Python云原生

构建应对海量用户数据的高可扩展Web应用

Cloud Native Python

[印] Manish Sethi 著
宋净超 译

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书以一个应用开发贯穿始终，从云原生和微服务的概念原理讲起，使用 Python 构建云原生应用，并使用 React 构建 Web 视图。为了应对大规模的互联网流量，使用了 Flux 构建 UI 和事件溯源及 CQRS 模式。考虑到 Web 应用的安全性，本书对此也给出了解决方案。书中对于关键步骤进行了详细讲解并给出运行结果。读者可以利用 Docker 容器、CI/CD 工具，敏捷构建和发布本书示例中的应用到 AWS、Azure 这样的公有云平台上，再利用平台工具对基础设施和应用的运行进行持续监控。

本书适合全栈工程师和想要使用 Python 构建云原生应用的开发者学习。

版权贸易合同登记号　图字：01-2017-6967

图书在版编目（CIP）数据

Python 云原生：构建应对海量用户数据的高可扩展 Web 应用 /（印）马尼什•塞西（Manish Sethi）著；宋净超译. —北京：电子工业出版社，2018.7

书名原文：Cloud Native Python

ISBN 978-7-121-34177-9

Ⅰ. ①P…　Ⅱ. ①马…　②宋…　Ⅲ. ①网页制作工具－程序设计　Ⅳ. ①TP393.092

中国版本图书馆 CIP 数据核字(2018)第 099661 号

策划编辑：孙奇俏
责任编辑：牛　勇
印　　刷：
装　　订：
出版发行：电子工业出版社
　　　　　北京市海淀区万寿路 173 信箱　　邮编 100036
开　　本：787×980　1/16　印张：20.25　字数：405 千字
版　　次：2018 年 7 月第 1 版
印　　次：2018 年 7 月第 1 次印刷
定　　价：89.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

译者序

本书主要讲解如何使用 Python 来构建云原生应用，其中包含了对云原生应用架构和微服务概念的解析，还包括了使用 React 来构建 Web UI，使用 MongoDB 来存储数据，使用 Kafka 消息队列、CQRS 和事件溯源的方式来支持分布式处理等内容。然后详细演示了如何使用 Jenkins 来做 CI/CD，将应用部署到 AWS 或 Azure 云平台上。

整本书一气呵成，一个示例贯穿始终，即构建一个“微博”应用。记得当年我还在读大学的时候，还使用 Java 构建过类似微博的 Web 应用，那也是我第一次构建 Web 应用，但是那时只是为了学习 JSP 和 Java Web 应用，也没有用到什么开发框架，而且只是在本地运行。我在看到本书时有种相见恨晚的感觉，如果当初我就读了这本书，那么使用 Python 构建一个可扩展的 Web 应用不是轻而易举吗？还可以将应用程序部署到云平台上，让自己的成果公之于众，这对于自己的技术生涯也是不小的激励。本书给出了构建应用的详细步骤和代码示例，甚至每一步的输出结果和页面截图都给出了，所以本书十分适合全栈工程师和想要使用 Python 构建云原生应用的开发者学习。

Python 也是我最喜欢的语言之一，它简单，优美，容易使用，而且是很多操作系统都内置支持的编程语言。Python 有着众多第三方包和框架，使用它不仅可以快速开发 Web 应用，还可以进行数据分析、机器学习，甚至调用其他语言，所以有人将 Python 称为“胶水”语言。现在使用 Python 构建云原生应用又成了广大 Python 爱好者的新方向。

在翻译本书前，我已经翻译过 *Cloud Native Go* 这本书，同时在很多会议和网站上分享过有关云原生的话题。还创建了“云原生应用架构”公众号（CloudNativeGo），欢迎读者朋友多多关注。也欢迎大家通过我的 Twitter（@jimmysongio）、GitHub（https://github.com/rootsongjc）和博客（https://jimmysong.io）与我交流。由于译者的精力和时间有限，书中难免会出现一些纰漏，欢迎广大读者指正。

Kubernetes 与云原生应用布道者　宋净超

2018 年 5 月于北京

推荐序

2000 年左右是“.com”的繁荣时代，那时我使用 C ++和 Perl 开发 Web 应用程序。那个年代人们必须亲自去 ISP 数据中心安装机器并进行 RAID 配置。2003—2006 年，应用开发转向了依靠基于虚拟机的共享主机。而如今，有了像 AWS、Azure、Google Cloud 这样的云计算提供商，以及 Python、Ruby 和 Scala 等编程语言，使得创建和扩展网站像玩玩具一样简单。

虽然云计算的出现使创建网络应用变得更简单，但是云计算的新工具、新部署方法和新工作流程不断出现，增加了学习云计算的难度。举个例子，开发人员应该使用哪些计算产品呢？软件即服务，平台即服务，还是基础架构即服务平台？开发人员应该选择 Docker 还是普通的虚拟机来部署？整个软件架构应该遵循 MVC 还是微服务模型？

本书作者 Manish 针对 Python 开发人员，全面讲解了云计算领域的各项技术。本书从云计算的分类及其产品基础开始讲起。书中的大多数章节都是独立的，读者可以选择阅读自己感兴趣的部分。本书简单明了地解释了像 CI 和 Docker 这样的复杂技术，满足了软件开发的敏捷模型所要求的，开发人员在几天（而不是几周）内学会使用新工具。本书给出了安装、配置的流程，并辅以代码，以使开发人员快速掌握所需的知识，从而提高工作效率。

本书尤其适合全栈开发者阅读，同时也适合新手和中级 Python 开发人员学习。本书旨在帮助 Python 开发者快速掌握当今软件开发所需要使用的工具和技术。

云计算的复杂性在于细节，无论是部署流程，管理基础设施，保护安全，还是建立工具生态系统，这些选择都将对正在构建软件应用的开发及运维团队产生深远的影响。

Ankur Gupta
NumerateLabs LLP 创始人
ImportPython & DjangoWeekly 主编

关于作者

Manish Sethi 是一名在印度班加罗尔工作的工程师。在他的职业生涯中，曾为初创公司和财富 10 强公司工作，帮助企业采用云原生方法来构建大规模可扩展产品。

他经常花时间学习和使用新技术，并积极地使用无服务器架构、机器学习和深度学习等方法解决实际问题。他还撰写博客，在聚会上发表演讲，从而为班加罗尔 DevOps 和 Docker 社区贡献自己的所学。

我要感谢我的兄弟 Mohit Sethi 和我的母亲 Neelam Sethi，在我的职业生涯中和编写本书的过程中，他们给了我非常多的鼓励和支持。

关于审校者

Sanjeev Kumar Jaiswal 是一名拥有 8 年行业经验的计算机工程师。他平时使用 Perl、Python 和 GNU/Linux 处理事务。他目前从事渗透测试、源代码评审、安全设计和实施以及 Web 和云安全项目开发相关工作。

目前，Sanjeev 也在学习 NodeJS 和 React Native。他喜欢教学，会教一些工程专业的学生和 IT 专业人员。在过去的 8 年中他一直利用业余时间教学。

2010 年，他秉承通过分享来学习的理想，为计算机科学专业的学生和 IT 专业人员建立了 Alien Coders（http://www.aliencoders.org）社区，这在印度的工程专业的学生中广受好评。可以通过 Facebook 主页 http://www.facebook.com/aliencoders、在 Twitter 上@aliencoders 以及 GitHub 页面 https://github.com/jassics 关注他。

他撰写了 *Instant PageSpeed Optimization* 一书，并与人共同撰写了 *Learning Django Web Development*，这两本书都已由 Packt 出版。他已为 Packt 审校了 7 本以上的书籍，并期待为 Packt 和其他出版商编写或审校更多书籍。

Mohit Sethi 是一名解决方案架构师，在云计算、存储、分布式系统、数据分析和机器学习等领域的 IaaS、PaaS 和 SaaS 方面，有超过 10 年的构建和管理产品的经验。此前，他曾在硅谷初创公司、财富 10 强公司和国防组织工作。他是一名有 12 年以上开源经验的贡献者，并且在班加罗尔举办 DevOps 聚会已经超过 3 年。

可以通过 Twitter（https://twitter.com/mohitsethi）、LinkedIn（https://in.linkedin.com/in/mohitsethi7）和 GitHub（https://github. com/mohitsethi）与他联系。

目录

前言

随着当今商业的迅速发展，企业为了支撑自身的迅速扩张，仅仅依靠自有的基础设施是远远不够的。因此，他们一直在追求利用云的弹性来构建支持高度可扩展应用程序的平台。

本书是你一站式地了解使用 Python 构建云原生应用架构的理想读本。本书首先介绍了什么是云原生应用架构以及它们能够帮助你解决哪些问题。然后介绍了如何使用 REST API 和 Python 构建微服务，并通过事件驱动的方式构建 Web 层。接下来，探讨了如何与数据服务进行交互，并使用 React 构建 Web 视图。之后详细介绍了应用程序的安全性和性能，以及如何在 Docker 中容器化你的服务。最后，讨论了如何在 AWS 和 Azure 平台上部署你的应用程序。在部署了应用程序后，围绕应用程序故障排查的一系列概念和技术结束了这本书。

本书内容

第 1 章　云原生应用和微服务简介，讨论云原生架构的基本概念和构建应用程序开发环境的方法。

第 2 章　使用 Python 构建微服务，构建自己的微服务知识体系并根据用例进行扩展。

第 3 章　使用 Python 构建 Web 应用，构建一个初始的 Web 应用程序并与微服务集成。

第 4 章　与数据服务交互，教你如何将应用程序迁移到不同的数据库服务。

第 5 章　使用 React 构建 Web 视图，讨论如何使用 React 构建用户界面。

第 6 章　使用 Flux 来构建 UI 以应对大规模流量，帮助你理解如何使用 Flux 创建可扩展的应用程序。

第 7 章　事件溯源与 CQRS，讨论如何以事件形式存储合约（transaction）。

第 8 章　Web 应用的安全性，讨论如何让你的应用程序免受外部威胁。

第 9 章　持续交付，介绍应用程序频繁发布的相关知识。

第 10 章　应用容器化，讨论容器服务和在 Docker 中运行应用程序的方法。

第 11 章　部署到 AWS 云平台，教你如何在 AWS 上构建基础设施并建立应用程序的生产环境。

第 12 章　部署到 Azure 云平台，讨论如何在 Azure 上构建基础设施并建立应用程序的生产环境。

第 13 章　监控云应用，介绍不同的基础设施和应用的监控工具。

阅读准备

你需要在系统上安装 Python，和一个文本编辑器，最好是 Vim、Sublime 或者 Notepad++。你需要下载 POSTMAN，这是一个功能强大的 API 测试套件，可以作为 Chrome 扩展插件来安装，可以从这里下载：`https://chrome.google.com/webstore/detail/postman/fhbjgbiflinjbdggehcddcbncdddomop?hl=en`。

除此之外，如果你还有如下 Web 应用的账号那就更好了：

- Jenkins
- Docker
- Amazon Web Services
- Terraform

如果没有以上账号，本书将指导你创建这些 Web 应用账号。

目标读者

本书适用于具有一些 Python 基础、熟悉命令行和 HTTP 应用程序基本原理的开发人员。对于那些想要了解如何构建、测试和扩展 Python 应用程序的人员来说，本书是一个理想选择。不需要有使用 Python 构建微服务的经验。

排版约定

在本书中，你会发现有许多类型的文本样式用于区分不同类型的信息。以下是这些样式的例子和含义解释。

文中的代码、数据表、目录名、文件名、文件扩展名、路径名、虚拟 URL、用户输入和 Twitter 将会如下显示："创建一个 `signup` 路由，其将采用 `GET` 和 `POST` 方法来读取页面，并将数据提交到后端数据库。"

代码块排版如下：

```
sendTweet(event){
  event.preventDefault();
  this.props.sendTweet(this.refs.tweetTextArea.value);
  this.refs.tweetTextArea.value = '';
}
```

命令行输入/输出将按如下格式排版：

```
$apt-get install nodejs
```

新术语与**重要词汇**以粗体显示。你在屏幕上看到的字，例如，对于在菜单或对话框中出现的条目将这样来描述："单击 **Create user** 按钮，创建用户，并在其上附加策略。"



警告或者重要提醒。



提示和技巧。

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **下载资源**：本书如提供示例代码及资源文件，均可在 下载资源 处下载。
- **提交勘误**：您对书中内容的修改意见可在 提交勘误 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动**：在页面下方 读者评论 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/34177



1

云原生应用和微服务简介

我们开始吧！在开始构建应用程序之前，我们需要先回答以下几个问题：

- 什么是云计算？包括哪些类型？
- 什么是微服务？它的基本概念是什么？
- 有哪些最佳实践？

在开始编写应用程序之前，本章我们先来了解应用程序开发者和程序员应该理解的基本概念。

首先来了解一下系统的构建及其演变。

长久以来，人们一直在探索如何以更好的方式来构建程序框架。随着技术的发展和更新、更好技术的采用，IT 框架变得更加可靠，对客户（或者顾客）来说也更加高效，同时还能使工程师们更加愉悦地工作。

持续交付有助于我们将软件开发周期转移到生产中，能够让我们从不同的视角识别出软件中容易出错的部分。我们坚定这样的理念：认真考虑每次代码提交，因为每次提交都有可能最终发布到生产。

我们已经深刻体会到了利用 Web 来进行机器之间的交互带来的便利。虚拟化平台能够让我们自由安排和调整机器的大小，利用基础计算单元来提供大规模处理这些机器的方法。一些大型高效的云平台（如 Amazon、Azure 和 Google）已经接受了小组织拥有其服务的全部生命周期的观念。诸如**领域驱动设计**（**DDD**）、**持续交付**（**CD**）、按需请求虚拟化、基础设施自动化、小型自治组织和规模化系统等概念，这些特征可以有效地促进我们高效地开

发软件并将其投入到生产。而现在，微服务的概念已经崛起。这不是脱离实际开发而编造出来的；它是一种实际的模式，或者说是一种因为实用而崛起的模式。本书我们先从云计算的类型讲起，以说明如何创建、监控和推进微服务。

许多组织都发现，掌握了细粒度的微服务结构，就可以快速将其转化为程序，了解更多的最新进展。微服务能使我们从根本上更灵活地应对和解决各种选择，让我们迅速适应变化，快速作出反应。

云计算简介

在开始介绍微服务和云原生概念之前，我们先来了解一下什么是云计算。

云计算是一个宽泛的术语，它的范围十分宽广。每当一个领域发生巨大变革的时候，就会存在很多的炒作，云就是这样一个热点领域，许多商家将本来与其风马牛不相及的事物牵强附会。由于云存在广泛的施展空间，组织可以选择何时、何地和如何使用云计算。

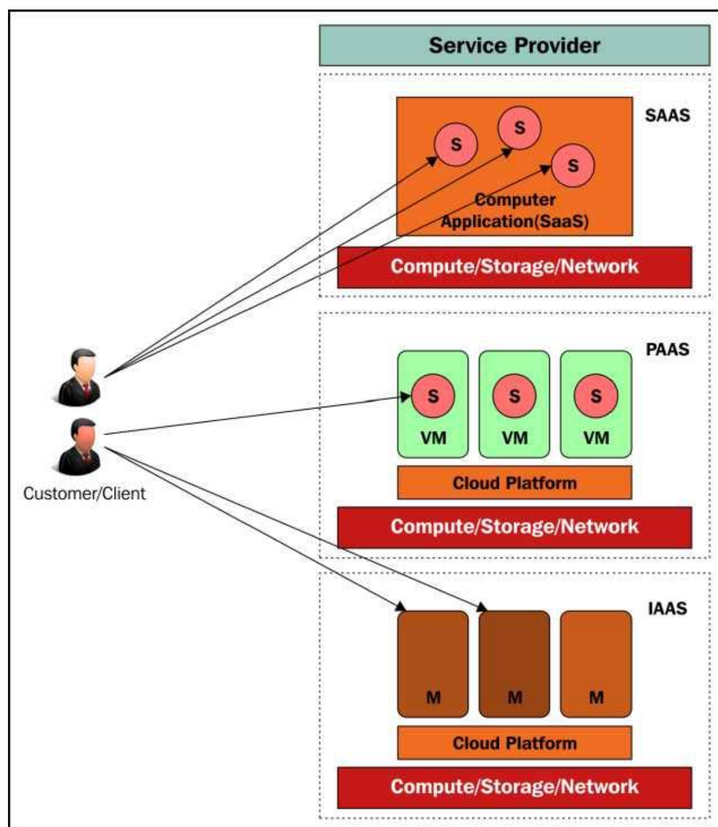
可以将云计算服务划分为以下几种类型。

- **SaaS**：终端用户可以直接使用的应用程序。
- **PaaS**：工具和服务的集合，其对于那些想要快速构建应用程序或快速将应用程序部署到生产环境，而不关心底层硬件的用户和开发者来说特别有用。
- **IaaS**：针对想要建立自己的商业模式并进行自定义的客户提供的类型。

云计算，作为一个技术栈，我们可以这样来解释它：

- 当我们谈及云计算时经常指的是一个技术栈，其涵盖了广泛的服务，其中很多服务是建立在其他服务的基础之上的，例如云。
- 云计算模式被认为是不同的可配置的计算资源的集合（例如服务器、数据库和存储），它们之间互相通信，几乎可以在无须人为监督的情况下提供给用户。

下图展示了云计算技术栈中包括的组件。



下面我们通过用例来详细了解云计算中的各个组件。

软件即服务

下面描述了 SaaS 的要点。

- **软件即服务（SaaS）**使用户能够通过浏览器访问由提供商提供的基于互联网提供服务的软件。这些服务是订阅式的，也被称为按需软件。
- 提供SaaS的公司产品包括Google Docs生产力套件、Oracle **CRM**（**客户关系管理**）、微软及其 Office365 产品，以及 Salesforce CRM 和 QuickBooks。
- SaaS 可以进一步细分为垂直 SaaS 和水平 SaaS，前者侧重于医疗保健和农业等特定行业的需求，后者专注于软件行业，例如人力资源和销售。
- SaaS 基本上是针对那些想要快速获取现成软件和应用的组织而言的，这些应用即

便是非技术人员也能理解和使用。组织基于对该软件的用途和预算，可选择不同的支持套餐。此外，你可以从世界的任何可以连接到互联网的地方访问这些服务。

平台即服务

下面描述了 PaaS 的要点。

- 对于 PaaS 产品，组织或企业无须担心其内部应用程序的硬件和软件基础设施的管理问题。
- PaaS 的最大优点是为开发团队（本地或远程）提供一些可能性，他们可以在通用框架上有效地构建、测试和部署应用程序，其中底层硬件和软件由 PaaS 服务提供商提供。
- PaaS 提供商提供平台，同时提供围绕该平台的服务。
- PaaS 提供商包括 Amazon Web Services（AWS Elastic Beanstalk）、微软 Azure（Azure Websites）、Google App Engine 和 Oracle（Big Data Cloud Service）。

基础设施即服务

下面描述了 IaaS 的要点。

- 与 SaaS 产品不同的是，IaaS 向客户提供的是 IT 资源，例如运行应用程序的裸机、用于存储的硬盘和用于联网的网络电缆，用户可以根据业务模式自定义。
- 对于 IaaS 产品，由于客户可以完全访问其基础设施，故他们可以根据其应用需求扩展 IT 资源。在 IaaS 产品中，客户必须管理应用程序/资源的安全性，并且在突发故障/崩溃时构建灾难恢复模型。
- 在 IaaS 中，服务是按需提供的，客户在使用时需要付费。因此，客户需要对资源进行成本分析，进行成本控制防止超出预算。
- 允许客户/消费者根据应用程序的要求定制其基础架构，在销毁基础架构后能够快速有效地重建。
- IaaS 服务基本上是按需付费的，这意味着你可以随时付款。IaaS 将根据你的资源使用情况和使用时间来收取费用。
- Amazon Web Services，提供**亚马逊弹性计算云（Amazon EC2）**和**亚马逊简易存储服务**（Amazon S3），是该云产品的第一大门户；然而，Microsoft Azure（虚拟机）、Rackspace（虚拟云服务器）和 Oracle（裸机云服务）也有这样的产品。

云原生概念

云原生是一种团队、文化和技术组织形式，利用自动化工具和架构来管理软件复杂度和加速软件交付。

云原生概念已经超出了技术的范畴。我们需要从成功的公司、团队和个人身上了解行业的发展方向。

目前,像 Facebook 和 Netflix 这样的大公司都已经在云原生技术上投入了大量的资源，而一些小的公司也意识到了该技术的价值。根据云原生技术实践的反馈，我们总结了如下一些优点。

- **结果导向和团队合作**：云原生的方法将一个大问题分解成众多的小问题，从而使每个团队都能专注于各自的部分。
- **减少重复工作**：自动化减少了困难、繁杂和重复的手动工作量，并减少了停机时间。这将使系统更有效率。
- **可靠而高效的应用程序基础设施**：自动化可以对不同环境（无论是开发、阶段发布还是生产）中的部署做更多控制，还可以处理意外事故。自动化构建不仅有助于正常部署，而且在灾难恢复的时候，也能使重新部署变得更加容易。
- **应用程序洞察**：为云原生应用程序构建的工具可为应用程序提供更多洞察，从而使调试、故障排查和审计变得容易。
- **高效可靠的安全性**：每个应用程序都会关注安全性，确保可靠的身份验证。云原生为开发人员提供了多种确保应用程序安全性的方式。
- **经济高效的系统**：使用云管理和部署应用程序可以有效地利用资源，包括应用程序发布，通过减少资源浪费使系统花费更合理。

云原生为何物？为何重要

云原生是一个宽泛的术语，它可以充分利用不同的技术，如基础架构自动化或中间件开发、支持服务等，这些功能都属于应用交付周期中的一部分。云原生方法包括频繁的版本发布（无 Bug、稳定），以及根据业务需求扩展应用程序。

使用云原生方法，可以系统地实现应用程序构建目标。

云原生方法远优于传统的面向虚拟化的业务流程，传统方法需要投入大量的精力来构

建开发环境，以及软件交付过程中的其他不同环境。理想的云原生架构应具有自动化和组合功能，这些自动化技术还应实现跨平台管理和部署应用程序。

云原生架构还应该具有其他几个特性，如稳定的日志记录、应用程序和基础架构监控，以确保应用程序正常运行。

云原生方法能够帮助开发人员使用例如 Docker 等工具，在不同的平台上轻易地创建和销毁应用程序。

云原生运行时环境

当软件从一个计算环境迁移到另一个计算环境时，该如何使其可靠运行？最佳解决方案就是使用容器。这可能是从开发者自己的机器到预发布环境再到生产环境，也可能是从物理机到私有云或公共云中的虚拟机。**Kubernetes** 已经成为容器服务的代名词，越来越流行。

随着云原生框架的兴起和其周边应用程序的增加，容器的编排越来越受人关注。在容器运行时需要注意以下几点。

- **管理容器状态和高可用性：**一定要保持容器的状态（创建和销毁），特别是在生产中，因为从业务的角度看它们非常重要。而且还应该能够根据业务需求进行扩展。
- **成本体现和分析：**容器可以根据业务预算控制资源管理，并可在很大程度上降低成本。
- **环境隔离：**在容器内运行的每个进程都被隔离在该容器内。
- **跨集群的负载均衡：**应用流量由容器集群处理，在容器内均匀重定向，这将增加应用程序最大响应数量并维持高可用性。
- **调试和灾难恢复：**由于我们要在生产系统中使用，所以需要确保有正确的工具可以监控应用程序的运行状况，采取必要措施避免停机，并提供高可用性。

云原生架构

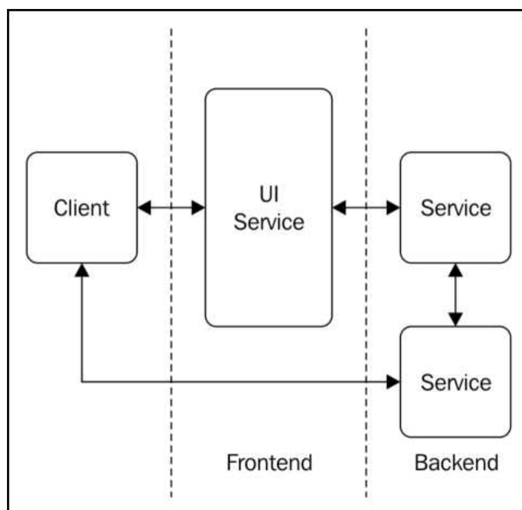
云原生架构类似于我们为遗留系统创建的应用程序架构，但在云原生应用程序架构中，我们应该考虑一些特性，例如十二要素应用程序（应用程序开发模式的集合）、微服务（将单体业务系统分解为独立可部署服务）、自服务敏捷基础设施（自服务平台）、基于 API 的协作（通过 API 进行服务之间的交互）和抗脆弱性（自我实现和加强的应用程序）。

首先，我们来探讨一下什么是微服务。

微服务是一个更宽泛的术语，指将大型应用程序分解成更小的模块，分别开发直到发布。这种方法不仅有助于有效地管理每个模块，而且还可以帮助我们发现服务底层本身的问题。以下是微服务的一些关键部分。

- **用户友好的界面**：微服务之间可以实现明确的分离。微服务的版本控制可以更好地对 API 进行控制，为消费者和生产者提供更大的自由度。
- **跨平台部署和 API 管理**：由于每个微服务都是一个单独的实体，因此可以更新单个微服务而不用更改其他服务，同时也更容易将服务回滚到之前的更改。这意味着用来部署微服务的工件应该兼容不同的 API 和数据模式。这些 API 必须在不同的平台上测试，并且测试结果应该在不同的团队，即运维、开发人员之间共享，大家共同维护一个集中的控制系统。
- **应用灵活性**：开发的微服务应该能够处理所有请求且必须做出响应，而不管请求的种类如何，包括可能的错误输入或无效请求。微服务也应该能够处理意外的负载请求并进行适当的响应。所以应当对微服务进行独立测试和集成测试。
- **微服务的分布**：最好将服务分为小块服务，以便单独跟踪和开发，最终组合起来形成一个微服务。这种技术使得微服务开发更稳定、更有效率。

下图显示了一个云原生应用程序的高级架构。

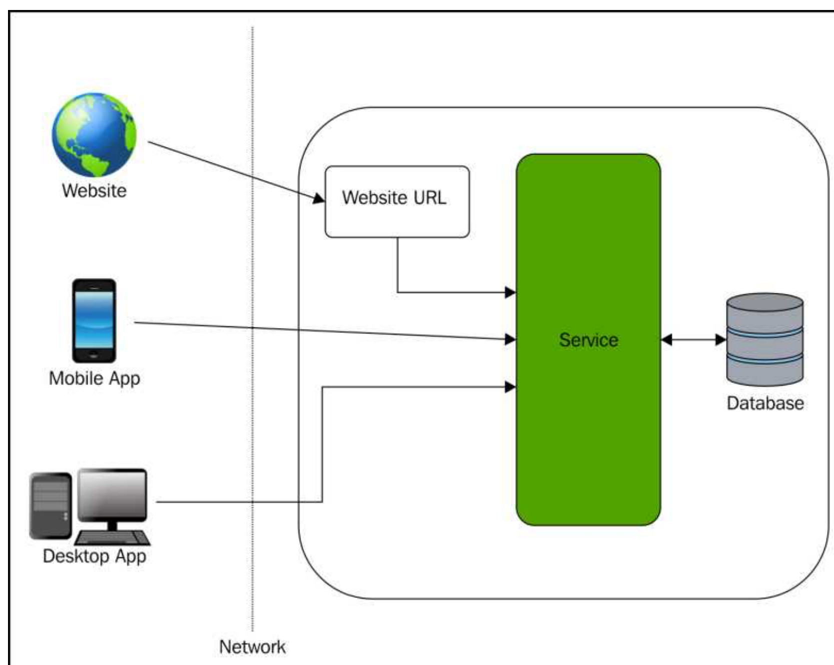


在理想情况下，应用程序体系结构应该从两三个服务开始，然后通过不断更新版本进行扩展。了解应用程序架构非常重要，因为应用程序需要与系统的不同组件集成，并且在大型组织中会有团队来专门管理这些组件。微服务中的版本控制至关重要，因为软件开发的各个阶段都会使用不同的方法。

微服务是一个新概念吗

微服务的概念由来已久了。微服务是一种用来划分大型系统中不同组件边界的架构模式。所有的微服务都以相似的方式工作，然后将不同服务链接起来，根据请求的类型处理特定事务的数据流。

下图描绘了微服务的体系结构。



为什么说 Python 是云原生微服务开发的最佳选择

为什么我们选择 Python 并将其推荐给其他人？下面列举了几点原因。

可读性

Python 是一种表达能力很强且易学习的编程语言。即便是业余爱好者也可以轻松掌握 Python。与其他编程语言（例如 Java 更关注圆括号、大括号、逗号和冒号）相比，使用 Python 你可以将更多的精力投入到编程上，减少调试语法的时间。

库和社区

Python 有大量可以在不同平台（例如 UNIX、Windows 和 OS X）上运行的库。这些库可以根据你的应用程序需求轻松扩展。同时还有一个强大的社区致力于构建这些库，这使得 Python 成为最适合用于业务的语言。

就 Python 社区而言，Python 用户组（PUG）是一个基于社区开发模式的社区，这可以促进 Python 在全球范畴内的普及。社区中的小组成员相互交流，有助于我们构建基于 Python 框架的大型系统。

交互模式

Python 交互模式可以帮助你在调试完代码后，立即将其添加到主程序中。

可扩展

Python 提供了更好的结构和概念，如模块，这比起任何其他脚本语言（如 shell 脚本），可以更系统地维护大型程序。

理解十二要素应用

一些实践经验更为云原生应用程序添光增彩。这些应用程序共同遵守**十二要素**应用程序宣言。该宣言概述了开发人员在构建现代 Web 应用程序时要遵循的方法。开发人员必须改变他们编码的方式，并为开发者和应用程序所运行的基础架构之间创建一个新的协议。

下面是在构建云原生应用程序时需要考虑的几个方面。

- 使用详实的设计，尽量自动化以降低时间成本和花费。
- 在不同环境（如阶段和生产）和不同平台（如 UNIX 或 Windows）中应用程序的可移植性。

- 使用适用于云平台的应用程序，并了解资源分配和管理。
- 使用一致的环境，减少持续交付/部署中的错误，从而最大限度地发挥软件的敏捷性。
- 通过最少的监督和设计灾难恢复架构来扩展应用程序，实现高可用性。

十二个要素中有许多要素是相互影响的。强调声明性配置，聚焦于速度、安全性和规模。十二要素应用程序的基本特征如下。

- **基准代码**：每份部署代码都使用版本控制追踪，并在不同的平台中部署多个实例。
- **依赖管理**：应用程序应该显式声明依赖关系，并使用工具来单独管理依赖，例如 Bundler、pip 和 Maven。
- **定义配置**：不同环境中的配置（例如环境变量）可能会不同，例如开发环境、预发布环境和生产环境应该在操作系统级定义。
- **后端服务**：所有资源都要被当作应用程序自身的一部分来对待。例如数据库、消息队列这样的后端服务应该被当作附加资源来看待，在所有的环境中以相同的方式被消费。
- **构建、发布、运行阶段隔离**：包括构建组件、绑定配置，根据绑定的组件和配置文件启动一个或多个实例。
- **进程无状态**：以一个或多个无状态进程运行应用（例如 master 和 worker），进程实例之间不共享任何内容。
- **服务端口绑定**：应用程序应当自包含，如果有任何需要对外暴露的服务，应当使用端口绑定的形式来完成（首选 HTTP）。
- **扩容无状态进程**：该架构应该强调基础平台中的无状态进程管理，而不是实现更复杂的应用程序。
- **进程状态管理**：进程应该可以迅速地增加，并在一小段时间后正常关闭。在这些方面可实现快速可扩展性、部署更改和灾难恢复。
- **持续发布和部署到生产**：保持环境一致，不论是预发布环境还是生产环境。这样可以保证在跨越不同的环境时获取相似的结果，有利于向生产环境持续交付。
- **把日志当作事件流**：不论是平台级的日志，还是应用级的日志，都十分重要，因为日志可以帮助你了解应用程序背后都做了什么。

- **后台管理任务被当作一次性进程运行**：在云原生的方法中，作为程序发布一部分的管理任务（例如数据库迁移）应该作为一次性进程运行，而不是作为常规应用程序长时间运行。

例如 Cloud Foundry、Heroku 和 Amazon Beanstalk 这些云平台都已为部署十二要素应用程序做了优化。

将应用程序与稳定的工程接口集成在一起，处理成无状态应用的设计，再考虑以上这些标准后，你的应用程序就可以部署到云平台了。Python 以其固有的、便捷的方式加速了 Web 应用程序的开发。

设置 Python 环境

全书都会用到 Python 环境，因此设置好开发环境是十分必要的，工欲善其事，必先利其器。

下面是本书后面部分需要用到的账号。

- 一个 GitHub 账号用于源代码管理。请参考链接中的文章创建：`https://medium.com/appliedcode/setup-github-account-9a5ec918bcc1`
- AWS 和 Azure 账号用于应用程序部署。请参考下面链接中的文章创建：
  - AWS `https://medium.com/appliedcode/setup-aws-account-1727ce89353e`
  - Azure `https://medium.com/appliedcode/setup-microsoft-azure-account-cbd635ebf14b`

现在，我们来设置开发环境需要用到的工具。

安装 Git

Git（`https://git-scm.com`）是一款免费的开源分布式版本控制系统，旨在快速高效地处理所有小型或大型项目。

在 Debian Linux 发行版（例如 Ubuntu）上安装 Git

在 Debian 系统上安装 Git 有很多种方式。

1. 使用**高级包管理工具（APT）**：你可以使用 APT 包管理工具更新本地包。然后，使用下面的命令以 root 用户身份执行、下载和安装最新版的 Git：

```
$ apt-get update –y
$ apt-get install git -y
```

执行以上命令将下载并安装 Git 到你的系统上。

2. 使用源码，可以按照下面的步骤来操作：

（1）从 GitHub 上下载源代码，从源码中编译软件。在开始前，我们先安装 Git 的依赖，使用 root 用户执行下面的命令：

```
$ apt-get update –y
$ apt-get install build-essential libssl-dev
libcurl4-gnutls-dev libexpatl-dev gettextunzip-y
```

（2）在安装好必要依赖后，可以去 Git 代码仓库（`https://github.com/git/git`）下载源代码，如下：

```
$ wget  https://github.eom/git/git/archive/vl.9.l.zip –O git.zip
```

（3）使用下面的命令解压下载的 ZIP 文件：

```
$ unzip git.zip
$ cd git-*
```

（4）需要编译包，使用 sudo 用户安装。执行下面的命令：

```
$ make prefix=/usr/local all
$ make prefix=/usr/local install
```

执行以上命令将在`/usr/local` 目录下安装 Git。

在 Debian 发行版系统上配置 Git

我们已经在系统上安装了 Git，现在还需要进行一些配置，以便让每次提交包含正确的

信息。

需要在配置中提供名字和电子邮件。可以使用以下命令添加这些值：

```
$ git config --global user.name "Manish Sethi"
$ git config --global user.email manish@sethis.in
```

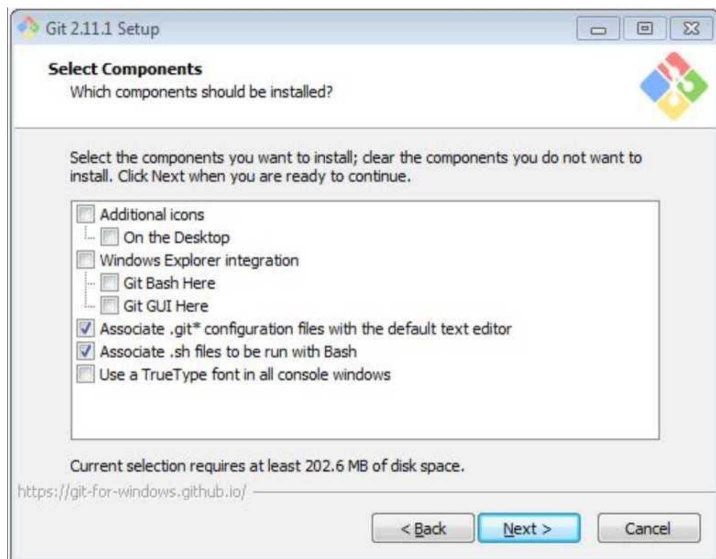
在 Windows 上安装 Git

可以从官方网站（`https://git-scm.com/download/win`）下载最新版本的 Git，按照以下步骤在 Windows 系统上安装 Git：

1. 下载了`.exe`文件后，双击运行。首先，你将会看到一个 GNU 证书，如下图所示。



单击 Next 按钮，如下图所示。

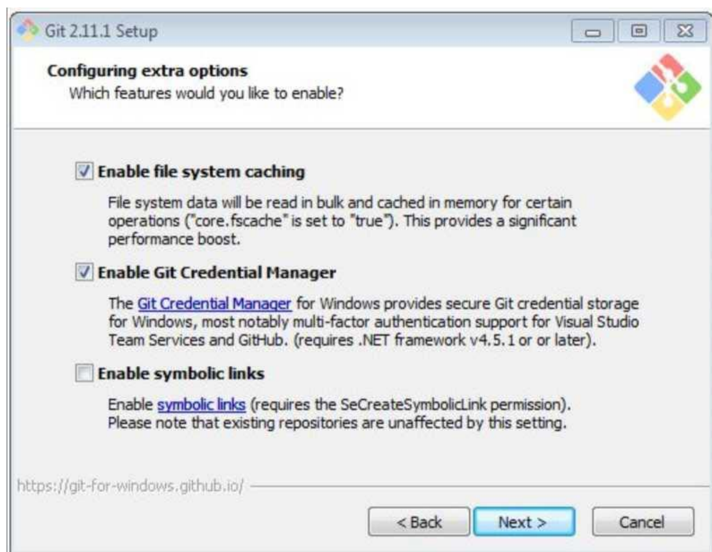


从该对话框可以看到，我们可以自定义要安装的程序，也可以使用默认配置，默认配置对于本书来说就可以了。

2. 另外，可以随 Git 同时安装 Git Bash。单击 **Next** 按钮，如下图所示。

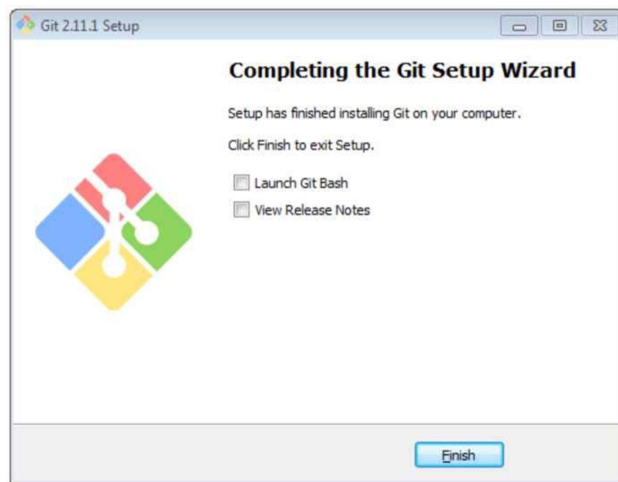


3. 在如下图所示的对话框中，可以启用 Git 包中的其他功能。然后单击 **Next** 按钮。



4. 可以单击 **Next** 按钮，跳过其余部分，完成安装。

完成安装后，将会看到如下图所示的界面。



至此，我们就在 Windows 上成功地安装了 Git。

使用 Chocolatey

笔者更青睐于使用这种方式在 Windows 10 上安装 Git。这种方式将安装与上面同样的包，但只需要一条命令。如果你从没听说过 Chocolatey，可以先去了解一下它。使用它安装软件仅需要一条命令，不需要在安装程序界面上点来点去！

Chocolatey 十分强大，我们可以将它和 **Boxstarter** 搭配使用来创建开发环境。如果你负责为 Windows 开发环境的人员设置机器，这种方法绝对值得一试。

我们来看看如何使用 Chocolatey 安装 Git。这里假定大家已经安装了 Chocolatey（`https://chocoiatey.org/install`，只要在命令提示符中输入一行命令即可）。

```
$ choco install git -params '"/GitAndUnixToolsOnPath"
```

执行该命令会安装 Git 和 BASH 工具并将它们加入到 Path 中。

在 Mac 上安装 Git

在开始安装 Git 之前，需要先给 OS X 系统安装命令行工具。

为 OS X 安装命令行工具

在安装任何开发者工具前，都需要先安装 Xcode（`https://developer.apple.com/xcode/`），这是一个大约 4GB 的开发者套件。在设置 GitHub 时需要用到的命令行工具（如 Git）就是 Xcode 开发者工具包中的一部分。苹果公司在 Mac App Store 中免费提供了这些工具。

如果你的电脑有足够的空间，那就下载安装 Xcode，因为这里面包括了完整的开发者工具集。

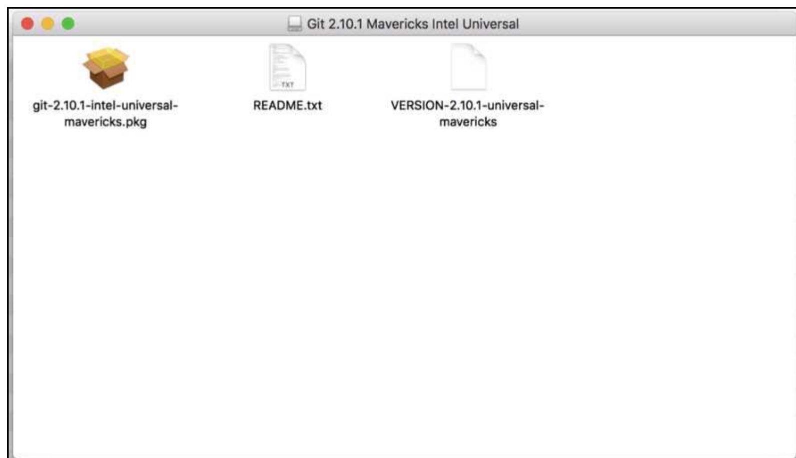
要下载命令行工具，需要在 `develop.apple.com` 上注册一个苹果开发者账号。创建完账号后，可以按照下列步骤选择命令行工具和 Xcode 的版本：

- 如果你使用的是 OS X 10.7.x，请下载 10.7 的命令行工具。如果你使用的是 OS X 10.8.x，请下载 10.8 的命令行工具。
- 下载完成后，打开 DMG 文件，根据提示步骤安装。

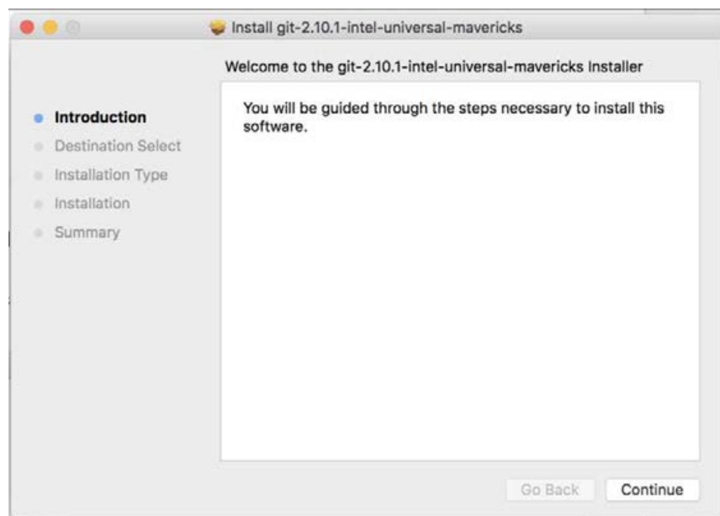
为 OS X 安装 Git

在 Mac 上安装 Git 就和在 Windows 上安装一样简单。不再使用`.exe`文件，而是使用`dmg`文件，可以从 Git 网站（`https://git-scm.com/download/mac`）下载。下面是安装步骤：

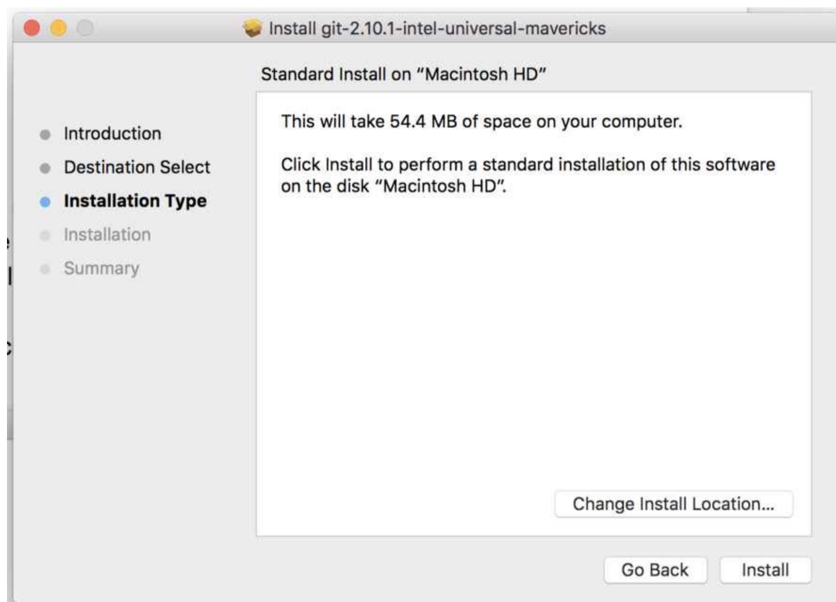
1. 双击下载的`dmg`文件，其会在 finder 中打开，如下图所示。



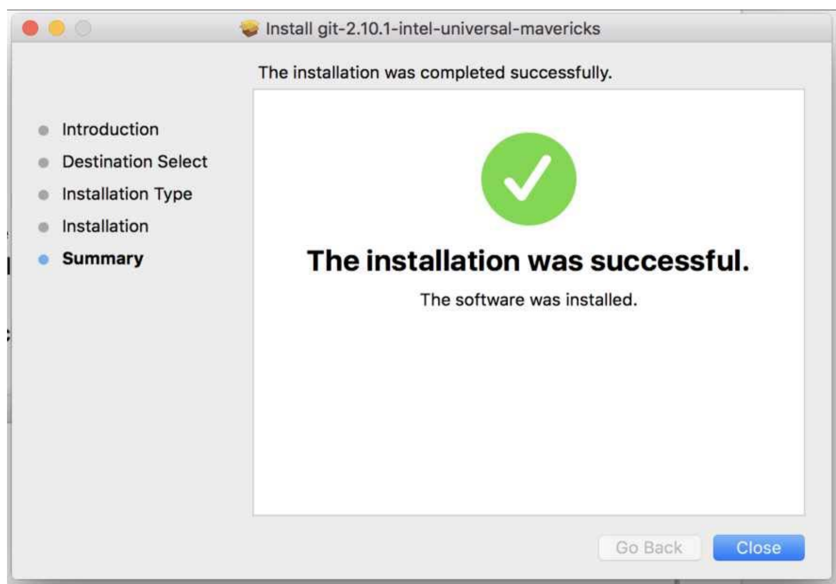
2. 双击包（图中的`git-2.10.1-intel-universal-mavericks.dmg`）文件，将会打开安装向导，如下图所示。



3. 单击 **Install** 按钮，如下图所示。



4. 安装完成后将会看到如下图所示的界面。





如果你使用的是 OS X 10.8，且没有修改安全性设置来运行安装第三方应用，则在安装该工具之前需要先调整 OS X 的设置。

安装和配置 Python

现在我们开始安装 Python，后面将用它来构建微服务。本书使用的是 Python 3.x 版本。

在 Debian 发行版（如 Ubuntu）上安装 Python

可以使用多种方式在 Debian 发行版上安装 Python。

使用 APT 包管理工具

可以使用 APT 包管理工具更新本地的包索引。然后使用 root 用户下载安装最新版本的 Python：

```
$ apt-get update –y
$ apt-get install python3 -y
```

下面的包将会被自动地下载和安装，因为它们是 Python 3 的必备组件：

```
libpython3-dev libpython3.4 libpython3.4-dev python3-chardet

python3-colorama python3-dev python3-distlib python3-html5lib

python3-requests python3-six python3-urllib3 python3-wheel python3.4-de
```

必备组件安装好了之后，将会自动下载和安装 Python。

使用源码安装

可以从 GitHub 上下载源码编译安装，步骤如下：

1. 开始安装前需要先安装 Git 的依赖。使用 root 用户执行下面的命令：

   ```
   $ apt-get update –y
   ```

```
$ apt-get install build-essential checkinstall libreadline-gplv2-
   dev libncursesw5-dev libssl-dev libsqlite3-dev tk-dev libgdbm-
   dev libc6-dev libbz2-dev -y
```

2. 从 Python 官方网站（`https://www.python.org`）下载 Python 安装包。可以下载到指定的位置：

```
$ cd/usr/local
$ wget https://www.python.Org/ftp/python/3.4.6/Python-3.4.6.tgz
```

3. 解压软件包：

```
$ tar xzf Python-3.4.6.tgz
```

4. 编译源码：

```
$ cd python-3.4.6
$ sudo ./configure
$ sudo make altinstall
```

5. 执行上面的命令将会在`/usr/local`下安装 Python。执行以下命令检查 Python 版本：

```
$ python3 –V
  python 3.4.6
```

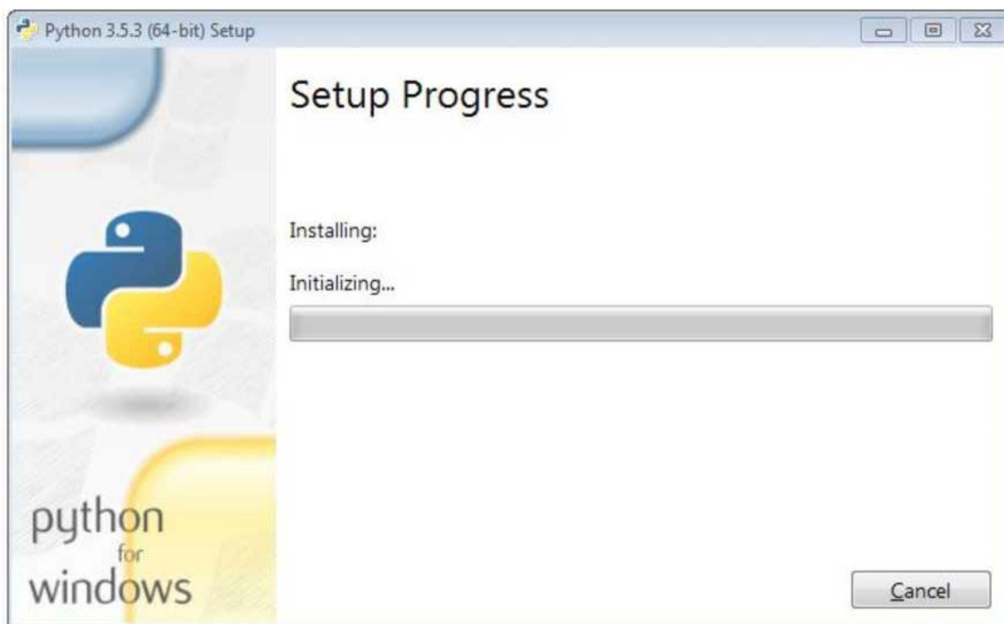
在 Windows 上安装 Python

下面我们来看如何在 Windows 7 和更新版本上安装 Python。在 Windows 上安装 Python 十分简单快捷。我们将使用 Python 3 以上版本，可以从 Python 官方网站（`https://www.python.org/downloads/windows`）下载。执行下面的步骤：

1. 根据你的系统配置下载 **Windows x86-64 executable installer** 安装程序。打开后开始安装，如下图所示。



2. 选择安装类型。单击 **Install Now** 选择默认配置安装，将会看到如下图所示的界面。



3. 安装完成后，将会看到如下图所示的界面。



到此就完成了 Windows 上的 Python 安装。

在 Mac 上安装 Python

在正式开始安装 Python 之前，需要先安装 OS X 命令行工具。如果你在安装 Git 的时候已经安装了命令行工具，可以忽略这一步。

为 OS X 安装命令行工具

在安装任何开发者工具前，都需要先安装 Xcode（`https://developer.apple.com/xcode`）；你需要到 `connect.apple.com` 上注册账号，然后下载对应版本的 Xcode 工具。

当然，还可以使用实用程序来安装命令行工具。随 Xcode 还安装了一个 `Xcode-select` 工具，如下所示：

```
% xcode-select --install
```

运行上述命令将会触发命令行工具的安装向导。按照安装向导的步骤就可以完成安装。

为 OS X 安装 Python

在 Mac 上安装 Python 和在 Windows 上安装 Git 一样简单。可以从 Python 官方网站（`https://www.python.org/downloads`）下载安装包。然后按照下列步骤安装：

1. 下载 Python 安装包后，双击开始**安装**，之后将会看到如下图所示的弹出窗口。

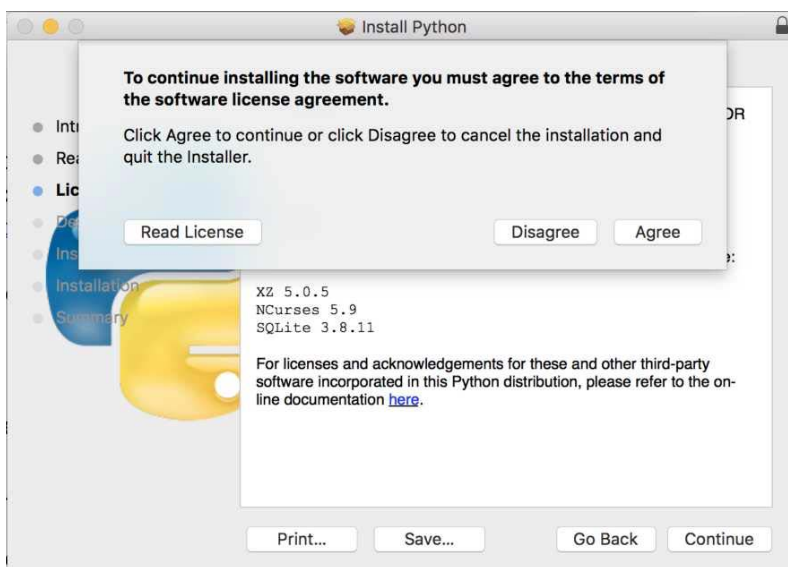


单击 **Continue** 按钮。

2. 显示 Python 的版本信息，如下图所示。

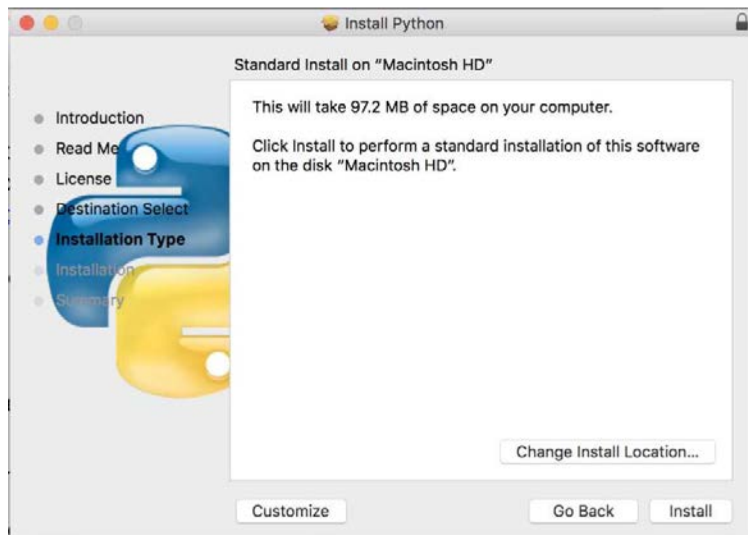


3. 单击 **Continue** 按钮。弹出证书，然后单击 **Agree** 按钮。这一步是必须的，如下图所示。

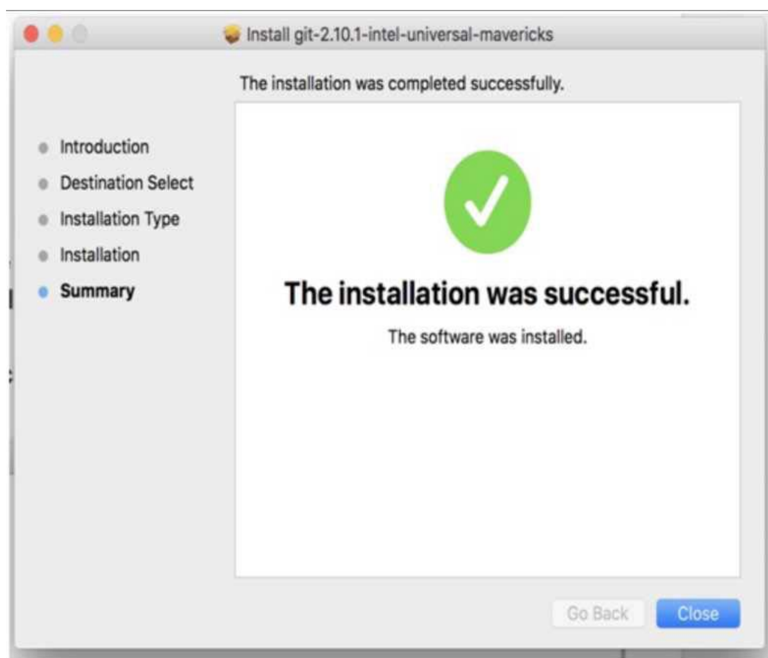


然后单击 **Continue** 按钮。

4. 提示安装信息，例如磁盘占用和安装路径（如下图所示）。单击 **Install** 按钮。



5. 安装完成后，将会看到如下图所示的界面。



6. 使用下面的命令查看安装的 Python 的版本：

```
% python3 –V
  Python 3.5.3
```

到此 Python 安装完成！

熟悉 GitHub 和 Git 命令

在本节中，我们将熟悉一些 Git 命令，这些命令在整本书中都会被用到。

- **git init**：初始化本地代码仓库。
- **git remote add origin <服务器>**：将本地目录<indexentry content="Git command:git remote add origin " dbid="164250"state="mod">链接到远程服务器的代码仓库，这样所有的更改都可以提交到远程仓库。
- **Git status**：列出所有已添加的文件/目录，或者是被修改和需要提交的文件/目录。
- **Git add * 或者 git add <文件名>**：添加文件/目录才能追踪本地文件<indexentry content="Git command:git add * or git add " dbid="164250" state="mod">，其才可以被提交。
- **git commit -m "提交信息"**：此命令可帮助你在本地计算机中提交跟踪的更改，并生成可识别更新代码的提交 ID。
- **Git commit -am "提交信息"**：和上一个命令唯一的不同就是，在将所有的文件添加到 stage 中后，该命令会根据操作系统打开一个默认的文本编辑器，例如 Ubuntu（使用 Vim）或 Windows（使用 Notepad++）来提交这些信息。
- **git push origin master**：将最后一次提交的代码推送到远程代码仓库。

测试以上命令确保环境安装正确。

至此，我们已经安装了 Git 和 Python，在后面构建微服务的时候会用到它们。在本节中，我们着重测试安装的软件并熟悉如何使用它们。

首先，我们先熟悉一下 Git 命令。它通过 HTTPS 获取外部 Python 代码（通常通过 GitHub），并将代码复制到本地的工作目录下：

```
$ git clone https://github.com/PacktPublishing/Cloud-Native-Python.git
```

执行上面的命令将会在本地机器上创建一个名为 `Cloud-Native-Python` 的目录。从系统的当前目录切换到 `Cloud-Native-Python/chapter1` 目录。

安装程序运行所需要的依赖包。在这里，我们只需要安装 Flask 模块：

```
$ cd hello.py
$ pip install requirements.txt
```

Flask 是用来做 Web 服务器的，在接下来的章节中将会详细讲解它。

安装完成后，使用下面的命令运行该应用：

```
$ python hello.py
* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

看到这样的输出大家应该感到很高兴吧！

```
$ curl http://0.0.0.0:5000/
Hello World!
```

如果看到上面的输出，证明你已经成功安装了 Python 环境。

下面让我们开始愉快的 Python 编程之旅吧！

本章小结

在本章中，我们首先了解了云平台和云计算技术栈，然后了解了十二要素应用程序方法论以及如何通过它们来进行微服务开发。最后按照步骤说明配置了我们自己的开发环境。

在下一章中，我们会使用 REST API 构建微服务，使用 Python 框架并测试 API 调用。

2

使用 Python 构建微服务

现在大家已经对微服务的基本概念和微服务的好处有所了解了，你一定迫不及待地想去构建微服务！本章就带领大家使用 REST API 构建微服务。

本章包括如下内容：

- 构建 REST API
- 测试构建的 API

Python 概念解析

我们先来了解一下 Python 的一些基础概念。

模块

模块与 Python 程序一样，能够让你基于逻辑来组织代码。而且只需要导入几行代码而不是引入整个 Python 程序。**模块**可以是一个或者多个函数类的组合。我们会用到很多 Python 库中的内置函数，需要的时候也可以编写自己的模块。

下面是一个模块结构的示例：

```
#myprogram.py
### EXAMPLE PYTHON MODULE
# Define some variables:
numberone = 1
```

```
age = 78

# define some functions
def printhello():
  print "hello"

def timesfour(input):
  print input * 4

# define a class
class house:
 def __init__(self):
      self.type = raw_input("What type of house? ")
      self.height = raw_input("What height (in feet)? ")
      self.price = raw_input("How much did it cost? ")
      self.age = raw_input("How old is it (in years)? ")

def print_details(self):
      print "This house is a/an " + self.height + " foot",
      print self.type, "house, " + self.age, "years old and costing\
      " + self.price + " dollars."
```

可以使用下面的命令导入上面定义的模块：

```
$ import myprogram
```

函数

函数是一个经过组织的、独立的程序块，用来执行特定的任务，可以将其嵌入更大的程序中来使用。函数的定义如下所示：

```
# function
def  functionname():
  do something
  return
```

需要记住以下几点：

- 缩进在 Python 程序中非常重要。

- 在默认情况下，函数的参数是有顺序的，需要按照顺序定义。

请看下面的代码片段：

```
def display ( name ):
#This prints a passed string into this function
  print ("Hello" + name)
  return;
```

可以这样来调用上面的函数：

```
display("Manish")
display("Mohit")
```

下图展示的是上面的程序运行的结果。

```
root@packtpub:/vagrant/github# cat function.py
def display ( name ):
     #This prints a passed string into this function
     print ("Hello" + name)
     return;


display(" Mohit ")
display(" Manish ")
root@packtpub:/vagrant/github# python function.py
Hello Mohit
Hello Manish
```



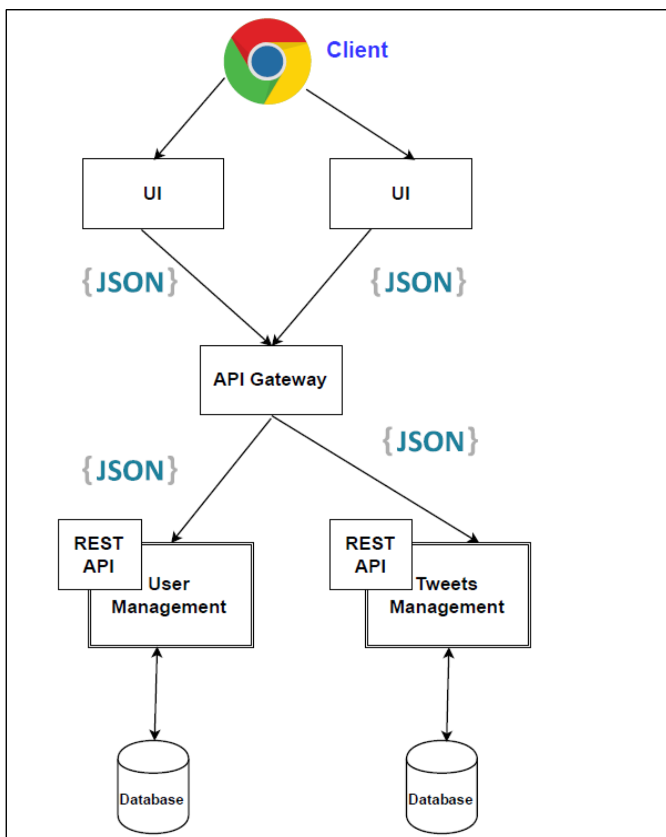
如果你的系统上安装了不止一个版本的 Python，那么默认使用 Python 2（通常是 2.7.x）。若要使用 Python 3 请使用 `python 3` 命令。

微服务模型

本书将开发一个可以独立运行的完整的 Web 应用。

通过前面章节的学习，我们已经对 Python 有了基本的了解，现在我们要开始对微服务进行建模，并了解应用程序的工作流程。

下图显示了微服务架构和应用程序的工作流程。



构建微服务

在本书中，我们将使用 **Flask** 作为 Web 框架来构建微服务。Flask 是一个十分强大且易用的 Web 框架。另外，我们还需要使用一些 Falsk 模板代码来创建一个简单的应用。

因为我们要根据十二要素应用的概念来构建应用，所以需要先创建一个集中管理的代码库。现在你应该已经有一个 GitHub 代码库了。如果没有的话，请参考第 1 章的介绍，创建一个 GitHub 账户，并将代码推送上去。

这里假设你已经创建了 GitHub 账户，我们将使用该代码库：`https://github.com/PacktPublishing/Cloud-Native-Python.git`。

现在我们设置本地目录用于下载远程代码库。为了确定当前位于 app 目录下，先执行下列命令：

```
$ mkdir Cloud-Native-Python  # Creating the directory
$ cd Cloud-Native-Python  # Changing the path to working directory
$ git init . # Initialising the local directory
$ echo "Cloud-Native-Python" > README.md  # Adding description of repository
$ git add README.md  # Adding README.md
$ git commit -am "Initial commit"  # Committing the changes
$ git remote add origin
https://github.com/PacktPublishing/Cloud-Native-Python.git  # Adding to
local repository
$ git push -u origin master  # Pushing changes to remote repository.
```

你将会看到如下输出。

```
root@packtpub:~# mkdir Cloud-Native-Python
root@packtpub:~# cd Cloud-Native-Python
root@packtpub:~/Cloud-Native-Python# git init .
Initialized empty Git repository in /root/Cloud-Native-Python/.git/
root@packtpub:~/Cloud-Native-Python# echo "Cloud-Native-Python" > README.md
root@packtpub:~/Cloud-Native-Python# git add README.md
root@packtpub:~/Cloud-Native-Python# git commit -am "Initial commit"
[master (root-commit) 3ff00d1] Initial commit
 1 file changed, 1 insertion(+)
 create mode 100644 README.md
root@packtpub:~/Cloud-Native-Python# git remote add origin  https://github.com/PacktPublishing/Cloud-Native-Python.git
root@packtpub:~/Cloud-Native-Python# git remote -v
origin  https://github.com/PacktPublishing/Cloud-Native-Python.git (fetch)
origin  https://github.com/PacktPublishing/Cloud-Native-Python.git (push)
```

第一次代码提交被成功地推送到了远程仓库。我们将继续按这种方式来构建应用程序，直到完成微服务构建。

现在我们需要安装一款基于文件的数据库 SQLite V3，用作微服务的数据存储。

使用下列命令安装 SQLite 3：

```
$ apt-get install sqlite3 libsqlite3-dev -y
```

创建和使用（source）`virtualenv` 环境，将本地应用程序的环境与全局 `site-package` 安装隔离开来。如果你的机器还未安装 `virtualenv`，则可以使用以下命令安装：

```
$ pip install virtualenv
```

使用下列命令创建 `virtualenv`：

```
$ virtualenv env --no-site-packages --python=python3
$ source env/bin/activate
```

上述命令的运行结果如下图所示。

```
root@packtpub:/vagrant/github/flask-microservices-app# virtualenv env --no-site-packages --python=python3
Running virtualenv with interpreter /usr/bin/python3
Using base prefix '/usr'
New python executable in /vagrant/github/flask-microservices-app/env/bin/python3
Also creating executable in /vagrant/github/flask-microservices-app/env/bin/python
Installing setuptools, pip, wheel...done.
root@packtpub:/vagrant/github/flask-microservices-app#  source env/bin/activate
(env) root@packtpub:/vagrant/github/flask-microservices-app# python
Python 3.4.3 (default, Oct 14 2015, 20:28:29)
[GCC 4.8.4] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

安装完 virtualenv 后，还要为虚拟环境安装依赖。使用下列命令在 requirements.txt 中添加一条依赖声明：

```
$ echo "Flask==0.10.1" >>  requirements.txt
```

后面还会有更多依赖包的声明被添加到该文件中。

将依赖声明文件中的依赖包安装到虚拟环境中：

```
$ pip install -r requirements.txt
```

依赖安装完成后，创建一个 app.py 文件，包含以下内容：

```
from flask import Flask

app = Flask(__name__)

if __name__ == '__main__':
  app.run(host='0.0.0.0', port=5000, debug=True)
```

上述代码显示了使用 Flask 的应用的基本结构。该代码初始化了 Falsk 变量，运行在 5000 端口，可以从任何地方（0.0.0.0）访问。

尝试运行程序，查看程序是否工作正常。执行下面的命令：

```
$ python app.py
```

命令输出如下图所示。

```
(env) root@packtpub:/vagrant/github/flask-microservices-app# cat app.py
from flask import Flask

app = Flask(__name__)

if __name__ == "__main__":
   app.run(host='0.0.0.0', port=5000, debug=True)
(env) root@packtpub:/vagrant/github/flask-microservices-app# python app.py
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger pin code: 267-323-539
```

到现在为止，我们在开始构建 RESTful API 之前，需要先确定访问的服务根 URL 是什么，才能进一步决定子 URL。如下面的示例：

```
http://[hostname]/api/v1/
```

在本示例中我们使用的是本地主机，因此 `hostname` 应该是 `locolhost` 加端口，Flask 应用的默认端口是 `5000`。因此该示例应用的根 URL 应该如下：

```
http://localhost:5000/api/v1/
```

再确定需要操作的资源类型，哪些可以作为服务暴露出来。在这个例子中，我们创建了两种资源类型：users 和 tweets。

users 和 info 类型具有如下表所示的 HTTP 方法。

HTTP 方法	URI	动　作
GET	`http://localhost:5000/api/v1/info`	返回版本信息
GET	`http://localhost:5000/api/v1/users`	返回用户列表
GET	`http://localhost:5000/api/v1/users/[user_id]`	返回指定 `user_id` 的详细用户信息
POST	`http://localhost:5000/api/v1/users`	根据传入的对象在后台服务器中创建一个新的用户
DELETE	`http://localhost:5000/api/v1/users`	删除传入的 JSON 文件中指定用户名的用户信息
PUT	`http://localhost:5000/api/v1/users/[user_id]`	更新 JSON 对象中指定的 `user_id` 的用户信息

我们可以使用客户端对资源执行 add、remove、modify 等操作。

基于本章内容，我们将使用在前面已经安装好的基于文件的数据库 SQLite3。

现在开始创建第一个资源/api/v1/info，输出可用版本和发布信息。

在此之前,我们需要先创建一个 apirelease 表结构,其中包含 API 版本和发布信息。运行下列命令：

```
CREATE TABLE apirelease(
buildtime date,
version varchar(30) primary key,
links varchar2(30), methods varchar2(30));
```

表创建完成后，使用下列命令向 SQLite3 中增加第一个版本(v1)：

```
Insert into apirelease values ('2017-01-01 10:00:00', "v1",
"/api/v1/users", "get, post, put, delete");
```

在 app.py 中定义函数和路由/api/v1/info 用于处理对/api/v1/info 路径的 RESTful 调用。

```
from flask import jsonify
import json
import sqlite3
@app.route("/api/v1/info")
def home_index():
    conn = sqlite3.connect('mydb.db')
    print ("Opened database successfully");
    api_list=[]
    cursor = conn.execute("SELECT buildtime, version,
    methods, links from apirelease")
for row in cursor:
    api = {}
    api['version'] = row[0]
    api['buildtime'] = row[1]
    api['methods'] = row[2]
    api['links'] = row[3]
    api_list.append(api)
```

```
    conn.close()
    return jsonify({'api_version': api_list}), 200
```

现在我们已经有了一个路由并且添加了处理函数，可访问 `http://localhost:5000/api/v1/info` 进行 RESTful 调用，结果如下图所示。

```
root@packtpub:/vagrant/github/flask-microservices-app# curl http://localhost:5000/api/v1/info -v
* Hostname was NOT found in DNS cache
*   Trying ::1...
* connect to ::1 port 5000 failed: Connection refused
*   Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 5000 (#0)
> GET /api/v1/info HTTP/1.1
> User-Agent: curl/7.35.0
> Host: localhost:5000
> Accept: */*
>
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
< Content-Type: application/json
< Content-Length: 177
< Server: Werkzeug/0.11.15 Python/3.4.3
< Date: Mon, 27 Feb 2017 12:18:35 GMT
<
{
  "api_version": [
    {
      "buildtime": "2017-01-01 10:00:00",
      "links": "/api/v1/users",
      "methods": "get, post, put, delete",
      "version": "v1"
    }
  ]
* Closing connection 0
}root@packtpub:/vagrant/github/flask-microservices-app#
```

如图所示，程序可以成功运行！

我们再来看`/api/v1/users`资源，通过它可以对用户记录执行多种操作。

为用户定义如下字段。

- `id`：用户的唯一标识（数字类型）。
- `username`：用户身份验证的唯一标识（字符串类型）。
- `emailid`：用户的邮箱（字符串类型）。
- `password`：用户的密码（字符串类型）。
- `full_name`：用户的全名（字符串类型）。

使用下列命令在 SQLite 中创建用户表结构：

```
CREATE TABLE users(
username varchar2(30),
emailid varchar2(30),
password varchar2(30), full_name varchar(30),
id integer primary key autoincrement);
```

构建 user 资源的方法

为 user 资源定义 GET 方法。

GET /api/v1/users

GET/api/v1/users 方法获取用户列表。通过在 app.py 中添加如下代码片段来创建/api/v1/users 路由：

```
@app.route('/api/v1/users', methods=['GET'])
def get_users():
  return list_users()
```

添加了路由后，定义 list_users()函数，连接数据库以获取用户列表。在 app.py 中添加如下代码：

```
def list_users():
conn = sqlite3.connect('mydb.db')
print ("Opened database successfully");
api_list=[]
cursor = conn.execute("SELECT username, full_name,
email, password, id from users")
for row in cursor:
a_dict = {}
a_dict['username'] = row[0]
a_dict['name'] = row[1]
a_dict['email'] = row[2]
a_dict['password'] = row[3]
a_dict['id'] = row[4]
api_list.append(a_dict)
```

```
conn.close()
  return jsonify({'user_list': api_list})
```

添加了路由和处理函数后，可以访问 http://localhost:5000/api/v1/users 来进行测试，如下图所示。

```
root@packtpub:/vagrant/github/flask-microservices-app# curl http://localhost:5000/api/v1/users
{
  "user_list": [
    {
      "email": "manishest@gmail.com",
      "id": 1,
      "name": "Manish",
      "password": "manish123",
      "username": "manish123"
    }
  ]
}root@packtpub:/vagrant/github/flask-microservices-app#
```

GET /api/v1/users/[user_id]

GET/api/v1/users/[user_id]获取指定 user_id 用户的详细信息。在 app.py 中为上述 GET 方法添加路由：

```
@app.route('/api/v1/users/<int:user_id>', methods=['GET'])
def get_user(user_id):
   return list_user(user_id)
```

从上面的代码我们可以看到，list_user(user_id)被路由到 list_user(user)函数，其还没有在 app.py 中定义。在 app.py 中定义获取指定用户详细信息的函数：

```
def list_user(user_id):
   conn = sqlite3.connect('mydb.db')
   print ("Opened database successfully");
   api_list=[]
   cursor=conn.cursor()
   cursor.execute("SELECT * from users where id=?",(user_id,))
   data = cursor.fetchall()
   if len(data) != 0:
       user = {}
```

```
        user['username'] = data[0][0]
        user['name'] = data[0][1]
        user['email'] = data[0][2]
        user['password'] = data[0][3]
        user['id'] = data[0][4]
            conn.close()
            return jsonify(a_dict)
```

在添加了 `list_user(user_id)`函数后，我们来测试一下，如下图所示。

```
root@packtpub:/vagrant/github/flask-microservices-app# curl http://localhost:5000/api/v1/users/2
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<title>404 Not Found</title>
<h1>Not Found</h1>
<p>The requested URL was not found on the server.  If you entered the URL manually please check your spelling and try again.</p>
root@packtpub:/vagrant/github/flask-microservices-app#
```

不好！ID 不存在！通常对于 ID 不存在的情况 Flask 应用会返回 404 错误。因为我们开发的是 Web 应用，因此需要为 API 请求返回 JSON 而不是 HTML。下面这段代码返回 404 错误：

```
from flask import make_response

@app.errorhandler(404)
def resource_not_found(error):
  return make_response(jsonify({'error':
  'Resource not found!'}),  404)
```

下图所示为运行结果。

```
root@packtpub:/vagrant/github/flask-microservices-app# curl http://localhost:5000/api/v1/users/2
{
  "error": "Resource not found!"
}root@packtpub:/vagrant/github/flask-microservices-app#
```

另外，还可以在 Flask 中添加 `abort` 库用于异常处理。同样，也可以为不同的 HTTP 错误码创建不同的错误处理方法。

现在 GET 方法已经成功运行,我们再来写一个 POST 方法,用于向列表中添加新用户。

向 POST 方法中传递数据有以下两种方法。

- **JSON：**使用这种方式，记录直接作为 JSON 结构体的一部分。RESTful API 调用如下：
  ```
  curl -i -H "Content-Type: application/json" -X POST -d
  {"field1":"value"} resource_url
  ```
- **参数形式：**使用这种方式，记录的值通过 URL 的参数传递：
  ```
  curl -i -H "Content-Type: application/json" -X POST
  resource_url?field1=val1&field2=val2
  ```

若使用 JSON，则输入的数据作为 JSON 结构体的一部分，同样要使用 JSON 的方式来读取这些数据。而在参数化的方法中，在 URL 中传递的数据（username 等）还要使用同样的方式来读取。

另外请注意，后端的 API 创建将随着 API 调用的类型而变化。

POST /api/v1/users

本书将采用第一种 POST 方式，即 JSON 方式。我们在 app.py 中为 POST 方法配置路由，然后调用更新数据库中的用户数据函数，如下：

```
@app.route('/api/v1/users', methods=['POST'])
def create_user():
  if not request.json or not 'username' in request.json or not
  'email' in request.json or not 'password' in request.json:
    abort(400)
user = {
    'username': request.json['username'],
    'email': request.json['email'],
    'name': request.json.get('name',""),
    'password': request.json['password']
}
  return jsonify({'status': add_user(user)}), 201
```

在上面的方法中我们调用了处理 400 错误代码的方法，下面实现这个方法：

```
@app.errorhandler(400)
def invalid_request(error):
   return make_response(jsonify({'error': 'Bad Request'}), 400)
```

定义 add_user(user)函数来更新用户记录。在 app.py 中进行如下定义：

```
def add_user(new_user):
 conn = sqlite3.connect('mydb.db')
 print ("Opened database successfully");
 api_list=[]
 cursor=conn.cursor()
 cursor.execute("SELECT * from users where username=? or
  emailid=?",(new_user['username'],new_user['email']))
data = cursor.fetchall()
if len(data) != 0:
     abort(409)
else:
     cursor.execute("insert into users (username, emailid, password,
full_name) values(?,?,?,?)",(new_user['username'],new_user['email'],
 new_user['password'], new_user['name']))
     conn.commit()
     return "Success"
 conn.close()
 return jsonify(a_dict)
```

添加 handler 和 POST 方法的路由后，使用 API 调用来测试：

```
curl -i -H "Content-Type: application/json" -X POST -d '{
"username":"mahesh@rocks", "email": "mahesh99@gmail.com",
"password": "mahesh123", "name":"Mahesh" }'
http://localhost:5000/api/v1/users
```

验证用户列表的 curl 请求，http://localhost:5000/api/v1/users，如下图所示。

```
root@packtpub:/vagrant/github/flask-microservices-app# curl -i -H "Content-Type: application/json" -X POST -d '{ "username":"mahesh@rocks", "email": "mahesh99@gmail.com", "password": "mahesh123", "name":"Mahesh" }' http://localhost:5000/api/v1/users
HTTP/1.0 201 CREATED
Content-Type: application/json
Content-Length: 25
Server: Werkzeug/0.11.15 Python/3.4.3
Date: Mon, 27 Feb 2017 11:15:03 GMT

{
  "status": "Success"
root@packtpub:/vagrant/github/flask-microservices-app# curl http://localhost:5000/api/v1/users
{
  "user_list": [
    {
      "email": "manishest@gmail.com",
      "id": 1,
      "name": "Manish",
      "password": "manish123",
      "username": "manish123"
    },
    {
      "email": "mahesh99@gmail.com",
      "id": 4,
      "name": "Mahesh",
      "password": "mahesh123",
      "username": "mahesh@rocks"
    }
  ]
}root@packtpub:/vagrant/github/flask-microservices-app#
```

DELETE /api/v1/users

delete 方法用于删除指定 username 的用户。需要传递一个 JSON 对象，其中必须包含 username 的 JSON 对象。

下面的代码片段在 app.py 中为用户创建了一个 DELETE 方法的路由：

```
@app.route('/api/v1/users', methods=['DELETE'])
def delete_user():
 if not request.json or not 'username' in request.json: abort(400)
 user=request.json['username']
  return jsonify({'status': del_user(user)}), 200
```

在下面的代码中，调用 del_user，验证 username 是否存在，若存在则删除 username 的记录。

```
def del_user(del_user):
  conn = sqlite3.connect('mydb.db')
  print ("Opened database successfully");
  cursor=conn.cursor()
  cursor.execute("SELECT * from users where username=? ",(del_user,))
  data = cursor.fetchall()
```

```
    print ("Data" ,data)
    if len(data) == 0:
      abort(404)
    else:
     cursor.execute("delete from users where username==?",
     (del_user,))
     conn.commit()
        return "Success"
```

我们给 DELETE 方法添加了/handler 路由，使用下面的 API 调用来测试：

```
    curl -i -H "Content-Type: application/json" -X delete -d '{
"username":"manish123" }' http://localhost:5000/api/v1/users
```

然后请求列出用户的 API（curl `http://localhost:5000/api/v1/users`），看看用户列表是否有变化，如下图所示。

```
root@packtpub:/vagrant/github/flask-microservices-app# curl -i -H "Content-Type: application/json" -X delete -d '{ "username":"manish123" }' http:
//localhost:5000/api/v1/users
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 25
Server: Werkzeug/0.11.15 Python/3.4.3
Date: Mon, 27 Feb 2017 11:23:13 GMT

{
  "status": "Success"
root@packtpub:/vagrant/github/flask-microservices-app# curl http://localhost:5000/api/v1/users
{
  "user_list": [
    {
      "email": "mahesh99@gmail.com",
      "id": 4,
      "name": "Mahesh",
      "password": "mahesh123",
      "username": "mahesh@rocks"
    }
  ]
}root@packtpub:/vagrant/github/flask-microservices-app#
```

用户已成功被删除！

PUT /api/v1/users

PUT API 用来更新指定 user_id 的用户记录。

在 app.py 文件中定义一个更新 user 记录的 PUT 方法的路由：

```
@app.route('/api/v1/users/<int:user_id>', methods=['PUT'])
def update_user(user_id):
  user = {}
```

```
    if not request.json:
        abort(400)
    user['id']=user_id
    key_list = request.json.keys()
    for i in key_list:
        user[i] = request.json[i]
    print (user)
    return jsonify({'status': upd_user(user)}), 200
```

定义 udp_user(user)函数，以更新 id 用户的数据：

```
def upd_user(user):
  conn = sqlite3.connect('mydb.db')
  print ("Opened database successfully");
  cursor=conn.cursor()
  cursor.execute("SELECT * from users where id=? ",(user['id'],))
  data = cursor.fetchall()
  print (data)
  if len(data) == 0:
    abort(404)
  else:
    key_list=user.keys()
    for i in key_list:
        if i != "id":
            print (user, i)
            # cursor.execute("UPDATE users set {0}=? where id=? ",
            (i, user[i], user['id']))
            cursor.execute("""UPDATE users SET {0} = ? WHERE id =
            ?""".format(i), (user[i], user['id']))
            conn.commit()
    return "Success"
```

添加了用户资源的 PUT 方法的 API 处理器后，我们来进行测试，如下图所示。

```
root@packtpub:/vagrant/github/flask-microservices-app# curl -i -H "Content-Type: application/json" -X put -d '{ "password":"mahesh@rocks" }' http://localhost:5000/api/v1/users/4
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 25
Server: Werkzeug/0.11.15 Python/3.4.3
Date: Mon, 27 Feb 2017 12:00:38 GMT

{
  "status": "Success"
root@packtpub:/vagrant/github/flask-microservices-app# curl http://localhost:5000/api/v1/users
{
  "user_list": [
    {
      "email": "mahesh99@gmail.com",
      "id": 4,
      "name": "Mahesh",
      "password": "mahesh@rocks",
      "username": "mahesh@rocks"
    }
  ]
}root@packtpub:/vagrant/github/flask-microservices-app#
```

我们之前定义的资源都属于 v1 版本，现在我们再定义 v2 版本的资源，向微服务中添加 tweet（推文）资源。在 users 资源中定义的用户可以对他们的推文进行操作。/api/info 如下图所示。

```
root@packtpub:/vagrant/github/flask-microservices-app# curl http://localhost:5000/api/v1/info -v
* Hostname was NOT found in DNS cache
*   Trying ::1...
* connect to ::1 port 5000 failed: Connection refused
*   Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 5000 (#0)
> GET /api/v1/info HTTP/1.1
> User-Agent: curl/7.35.0
> Host: localhost:5000
> Accept: */*
>
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
< Content-Type: application/json
< Content-Length: 317
< Server: Werkzeug/0.11.15 Python/3.4.3
< Date: Mon, 27 Feb 2017 12:16:54 GMT
<
{
  "api_version": [
    {
      "buildtime": "2017-01-01 10:00:00",
      "links": "/api/v1/users",
      "methods": "get, post, put, delete",
      "version": "v1"
    },
    {
      "buildtime": "2017-01-11 12:20:00",
      "links": "/api/v2/tweets",
      "methods": "get, post",
      "version": "v2"
    }
  ]
* Closing connection 0
}root@packtpub:/vagrant/github/flask-microservices-app#
```

推文资源的 HTTP 方法如下表所示。

HTTP 方法	URI	动　　作
GET	http://localhost:5000/api/v2/tweets	获取推文列表
GET	http://localhost:5000/api/v2/users/[user_id]	获取指定 ID 的推文
POST	http://localhost:5000/api/v2/tweets	发送新的推文并保存到数据库中

推文中包括如下字段。

- `id`：每条推文的唯一标识（数字类型）。
- `username`：应该是 users 资源中存在的用户（字符串类型）。
- `body`：推文的内容（字符串类型）。
- `Tweet_time`：（指定类型）。

在 SQLite 3 中定义推文资源表结构：

```
CREATE TABLE tweets(
id integer primary key autoincrement,
username varchar2(30),
body varchar2(30),
tweet_time date);
```

推文资源的表结构创建好后，就可以开始创建获取推文资源的 `GET` 方法了。

构建 tweet 资源的方法

下面将创建对数据库中推文执行不同操作的方法和对应的 API。

GET /api/v2/tweets

该方法可以获取所有用户的所有推文。

在 `app.py` 中添加如下代码指定 `GET` 方法的路由：

```
@app.route('/api/v2/tweets', methods=['GET'])
def get_tweets():
  return list_tweets()
  Let's define0020list_tweets() function which connects to database and
get us all the tweets and respond back with tweets list
def list_tweets():
```

```
        conn = sqlite3.connect('mydb.db')
        print ("Opened database successfully");
        api_list=[]
        cursor = conn.execute("SELECT username, body, tweet_time, id from
  tweets")
      data = cursor.fetchall()
      if data != 0:
          for row in cursor:
              tweets = {}
              tweets['Tweet By'] = row[0]
              tweets['Body'] = row[1]
              tweets['Timestamp'] = row[2]
      tweets['id'] = row[3]
              api_list.append(tweets)
      else:
          return api_list
      conn.close()
      return jsonify({'tweets_list': api_list})
```

在添加了获取所有推文的方法后，通过 RESTful API 调用来测试上述代码，如下图所示。

```
root@packtpub:/vagrant/github/flask-microservices-app# curl http://localhost:5000/api/v2/tweets -v
* Hostname was NOT found in DNS cache
*   Trying ::1...
* connect to ::1 port 5000 failed: Connection refused
*   Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 5000 (#0)
> GET /api/v2/tweets HTTP/1.1
> User-Agent: curl/7.35.0
> Host: localhost:5000
> Accept: */*
>
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
< Content-Type: application/json
< Content-Length: 23
< Server: Werkzeug/0.11.15 Python/3.4.3
< Date: Mon, 27 Feb 2017 12:36:56 GMT
<
{
  "tweets_list": []
* Closing connection 0
}root@packtpub:/vagrant/github/flask-microservices-app#
```

因为到目前为止我们还没有添加任何推文，所以返回的结果是空的。下面我们来添加几条推文试一试。

POST /api/v2/tweets

POST 方法为指定用户添加推文。

下面代码在 app.py 中添加 POST 方法的路由：

```
@app.route('/api/v2/tweets', methods=['POST'])
def add_tweets():
  user_tweet = {}
  if not request.json or not 'username' in request.json or not
 'body' in request.json:
    abort(400)
user_tweet['username'] = request.json['username']
user_tweet['body'] = request.json['body']
user_tweet['created_at']=strftime("%Y-%m-%dT%H:%M:%SZ", gmtime())
print (user_tweet)
return  jsonify({'status': add_tweet(user_tweet)}), 200
```

定义为指定用户添加推文的 add_tweet(user_tweet)方法：

```
def add_tweet(new_tweets):
  conn = sqlite3.connect('mydb.db')
  print ("Opened database successfully");
  cursor=conn.cursor()
  cursor.execute("SELECT * from users where username=? ",
(new_tweets['username'],))
data = cursor.fetchall()

if len(data) == 0:
    abort(404)
else:
    cursor.execute("INSERT into tweets (username, body, tweet_time)
values(?,?,?)",(new_tweets['username'],new_tweets['body'],
new_tweets['created_at']))
    conn.commit()
    return "Success"
```

使用 RESTful API 调用测试上面定义的向数据库中添加推文的函数：

```
curl -i -H "Content-Type: application/json" -X POST -d '{
"username":"mahesh@rocks","body": "It works" }'
http://localhost:5000/api/v2/tweets
```

结果如下图所示。

```
root@packtpub:/vagrant/github/flask-microservices-app# curl -i -H "Content-Type: application/json" -X POST -d '{ "username":"mahesh@rocks","body"
 "It works" }' http://localhost:5000/api/v2/tweets
HTTP/1.0 201 CREATED
Content-Type: application/json
Content-Length: 25
Server: Werkzeug/0.11.15 Python/3.4.3
Date: Mon, 27 Feb 2017 12:40:53 GMT

{
  "status": "Success"
}root@packtpub:/vagrant/github/flask-microservices-app#
```

检查推文是否真的被成功添加到了数据库中：

```
curl http://localhost:5000/api/v2/tweets –v
```

结果如下图所示。

```
root@packtpub:/vagrant/github/flask-microservices-app# curl http://localhost:5000/api/v2/tweets -v
* Hostname was NOT found in DNS cache
*   Trying ::1...
* connect to ::1 port 5000 failed: Connection refused
*   Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 5000 (#0)
> GET /api/v2/tweets HTTP/1.1
> User-Agent: curl/7.35.0
> Host: localhost:5000
> Accept: */*
>
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
< Content-Type: application/json
< Content-Length: 156
< Server: Werkzeug/0.11.15 Python/3.4.3
< Date: Mon, 27 Feb 2017 13:02:38 GMT
<
{
  "tweets_list": [
    {
      "body": "It works",
      "id": 2,
      "timestamp": "2017-02-27T12:40:53Z",
      "tweetedby": "mahesh@rocks"
    }
  ]
* Closing connection 0
}root@packtpub:/vagrant/github/flask-microservices-app#
```

由上图可知，我们已经添加了第一条推文。那么如果只想看指定 ID 的推文该怎么办呢？可以在 GET 方法中指定 user_id。

GET /api/v2/tweets/[id]

GET 方法获取指定 ID 的推文。

在 app.py 中添加如下代码，定义获取指定 ID 的 GET 方法的路由：

```
@app.route('/api/v2/tweets/<int:id>', methods=['GET'])
def get_tweet(id):
  return list_tweet(id)
```

定义 list_tweet()方法，该方法用来连接数据库，获取指定 ID 的推文，返回 JSON 数据：

```
def list_tweet(user_id):
  print (user_id)
  conn = sqlite3.connect('mydb.db')
  print ("Opened database successfully");
  api_list=[]
 cursor=conn.cursor()
 cursor.execute("SELECT * from tweets  where id=?",(user_id,))
 data = cursor.fetchall()
 print (data)
 if len(data) == 0:
   abort(404)
 else:

   user = {}
   user['id'] = data[0][0]
   user['username'] = data[0][1]
   user['body'] = data[0][2]
   user['tweet_time'] = data[0][3]

conn.close()
return jsonify(user)
```

添加了获取指定 ID 的推文方法后，使用下面的 RESTful API 调用测试：

```
curl http://localhost:5000/api/v2/tweets/2
```

结果如下图所示。

```
root@packtpub:/vagrant/github/flask-microservices-app# curl http://localhost:5000/api/v2/tweets/2
{
  "body": "It works",
  "id": 2,
  "tweet_time": "2017-02-27T12:40:53Z",
  "username": "mahesh@rocks"
}root@packtpub:/vagrant/github/flask-microservices-app#
```

至此，我们已经成功构建了 RESTful API，作为访问数据所需的微服务，并对其执行了各种操作。

测试 RESTful API

到目前为止我们构建的所有 RESTful API 都是通过访问根 URL 来确认后端方法是否可以正确执行的。每次写了新的代码都要从头执行一遍来确认是否可以在生产环境上正确运行。下面，我们将编写可以作为系统单独运行的测试用例，以确保后端服务可以在生产环境正确运行。

有多种不同类型的测试。

- **功能性测试：**主要用于测试组件或系统的功能。对组件的功能规范执行此测试。
- **非功能性测试：**这种测试针对组件的质量特性进行测试，其中包括效率测试、可靠性测试等。
- **结构测试：**这种类型的测试用于测试系统的结构。为了编写测试用例，测试人员需要了解代码的内部实现。

在本节中，我们将根据上面的应用编写测试用例，特别是单元测试用例。我们将编写自动测试的 Python 代码，测试所有 API 调用，返回测试结果。

单元测试

单元测试用于测试系统中的工作单元或逻辑单元代码。以下是单元测试用例的特点。

- **自动化**：可以自动执行。
- **独立**：不应该有任何依赖。
- **持续可重复**：保持幂等性。
- **可维护**：容易理解和更新。

我们使用名为 nose 的单元测试框架，还可以使用 docstest（`https://docs.python.org/2/library/doctest.html`）来测试。

使用下面的命令安装 nose：

```
$ pip install nose
```

也可以把它写到 `requirement.txt` 文件中，执行下面的命令安装：

```
$ pip install -r requirements.txt
```

安装完 nose 测试框架后，在另一个名为 `flask_test.py` 的文件中初始化测试用例。

```
from app import app
import unittest

class FlaskappTests(unittest.TestCase):
  def setUp(self):
     # creates a test client
     self.app = app.test_client()
     # propagate the exceptions to the test client
     self.app.testing = True
```

上述代码可以初始化 `slef.app` 测试用例。

在 FlaskappTest 类中添加测试用例，获取 `GET /api/v1/users` 的返回码：

```
def test_users_status_code(self):
    # sends HTTP GET request to the application
    result = self.app.get('/api/v1/users')
    # assert the status code of the response
```

```
    self.assertEqual(result.status_code, 200)
```

上述代码将测试访问/api/v1/users 是否返回 200；如果没有将返回错误，表示测试失败。从代码中可以看到，这段代码没有依赖任何其他代码，这就是所谓的单元测试。

如何运行这段代码？我们已经安装了 nose 测试框架，只要在测试文件（本例中是 flask_test.py）所在的当前目录下执行下面的命令即可：

```
$ nosetests
```

以此类推，为本章前面创建的不同资源方法的 RESTful API 编写更多的测试用例。

- GET/api/v2/tweets 的测试用例如下：

```
def test_tweets_status_code(self):
  # sends HTTP GET request to the application
  result = self.app.get('/api/v2/tweets')
  # assert the status code of the response
  self.assertEqual(result.status_code, 200)
```

- GET/api/v1/info 的测试用例如下：

```
def test_tweets_status_code(self):
    # sends HTTP GET request to the application
    result = self.app.get('/api/v1/info')
    # assert the status code of the response
    self.assertEqual(result.status_code, 200)
```

- POST/api/v1/users 的测试用例如下：

```
def test_addusers_status_code(self):
    # sends HTTP POST request to the application
    result = self.app.post('/api/v1/users', data='{"username":
"manish21", "email":"manishtest@gmail.com", "password": "test123"}',
content_type='application/json')
    print (result)
    # assert the status code of the response
    self.assertEquals(result.status_code, 201)
```

- PUT/api/v1/users 的测试用例如下：

```
def test_updusers_status_code(self):
    # sends HTTP PUT request to the application
    # on the specified path
```

```
    result = self.app.put('/api/v1/users/4', data='{"password":
"testing123"}', content_type='application/json')
    # assert the status code of the response
    self.assertEquals(result.status_code, 200)
```

- `POST/api/v1/tweets` 的测试用例如下：

```
def test_addtweets_status_code(self):
    # sends HTTP GET request to the application
    # on the specified path
    result = self.app.post('/api/v2/tweets', data='{"username":
"mahesh@rocks", "body":"Wow! Is it working #testing"}',
content_type='application/json')

    # assert the status code of the response
    self.assertEqual(result.status_code, 201)
```

- `DELETE/api/v1/users` 的测试用例如下：

```
def test_delusers_status_code(self):
    # sends HTTP Delete request to the application
    result = self.app.delete('/api/v1/users', data='{"username":
"manish21"}', content_type='application/json')
    # assert the status code of the response
    self.assertEquals(result.status_code, 200)
```

同样，你还可以根据自己的想法编写更多的测试用例，使这些 RESTful API 更可靠和无 Bug。

执行上述所有测试用例，看看测试是否可以通过。`flask_test.py` 脚本的执行结果如下图所示。

```
(env) root@packtpub:/vagrant/github/flask-microservices-app# nosetests
......
----------------------------------------------------------------------
Ran 6 tests in 0.478s

OK
(env) root@packtpub:/vagrant/github/flask-microservices-app#
```

太棒了！所有的测试都通过了，我们可以进入下一阶段了，即为这些 RESTful API 创建 Web 页面。

本章小结

本章我们主要通过编程来构建微服务，由此我们对 RESTful API 的工作原理有所了解，还了解到如何扩展这些 API，并了解了这些 API 的 HTTP 响应。此外，我们还学习了如何编写测试用例，这对于确保代码在生产环境正确运行都是很重要的。

3

使用 Python 构建 Web 应用

在上一章中我们主要关注如何构建和测试微服务的后端 RESTful API。可以使用 curl 或测试框架，例如 nose、unittest2 等来测试。本章，我们将编写 HTML 页面、JavaScript REST 客户端来与微服务交互。

本章包括的主要内容有：

- 构建 HTML 页面和数据绑定
- 使用 knockout.js 的 JavaScript REST 客户端

在本章中，我们将创建一个客户端应用程序，该应用程序从 HTML 网页收集动态内容，根据用户的操作对后端服务的响应进行更新。

你肯定听说过许多采用 MVC 模式的应用程序框架，包括 MVC（Model View Controller）、MVP（Model View Presenter）和 MVVM（Model View ViewModel）等。

在本书的例子中，我们将使用 **knockout.js**，这是一个基于 MVVM 模式的 JavaScript 库，其可以帮助开发人员构建丰富的响应式网站。它可以独立工作也可以与其他 JavaScript 库一起使用，如 jQuery。Knockout.js 将 UI 与基础 JavaScript 模型绑定。这些模型根据 UI 的更改进行更新，反之亦然，也就是说是双向数据绑定。

下面我们讨论 knockout.js 中两个重要的概念：绑定和可观察。



knockout.js 是一个 JavaScript 库，通常用于开发桌面 Web 应用程序。它提供了与数据源同步的响应机制，还有数据模型和用户界面之间的双向绑定机制。可在下面位置阅读更多关于 knockout.js 资料：`http://knockoutjs.com/documentation/introduction.html`。

在本章中，我们将创建一个 Web 应用程序，将一个用户和推文添加到数据库中并验证。

应用入门

首先创建 HTML 模板。在应用程序的根目录下创建一个 `template` 目录，我们将在该目录下创建所有的功能模板。

首先创建 `adduser.html` 文件的基本框架，如下所示：

```
<!DOCTYPE html>
<html>
  <head>
    <title>Tweet Application</title>
  </head>
  <body>
    <div class="navbar">
     <div class="navbar-inner">
        <a class="brand" href="#">Tweet App Demo</a>
     </div>
    </div>
   <div id="main" class="container">

    Main content here!
   </div>
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link href="http://netdna.bootstrapcdn.com/twitter-
   bootstrap/2.3.2/css/bootstrap-combined.min.css" rel="stylesheet">
  <script src="http://ajax.aspnetcdn.com/ajax/jquery/jquery-
    1.9.0.js"></script>
```

```
    <script src="http://netdna.bootstrapcdn.com/twitter-
      bootstrap/2.3.2/js/bootstrap.min.js"></script>
    <script src="http://ajax.aspnetcdn.com/ajax/knockout/knockout-
      2.2.1.js"></script>
    </body>
</html>
```

正如你在代码中看到的，我们指定了一些用于处理 HTML 响应的`.js`脚本。这类似于 twitter-bootstrap，其具有`<meta name="viewport">`属性，以帮助根据浏览器维度扩展页面。

创建应用程序用户

在开始编写 Web 页面前，需要先设置创建用户的路由，如下：

```
from flask import render_template

@app.route('/adduser')
def adduser():
  return render_template('adduser.html')
```

创建了路由后，在`adduser.html`中创建一个表单，用于获取用户信息并提交：

```
<html>
   <head>
      <title>Twitter Application</title>
   </head>
   <body>
    <form>
      <div class="navbar">
       <div class="navbar-inner">
          <a class="brand" href="#">Tweet App Demo</a>
       </div>
    </div>
    <div id="main" class="container">

     <table class="table table-striped">
```

```
        Name: <input placeholder="Full Name of user" type "text"/>
        </div>
        <div>
          Username: <input placeholder="Username" type="username">
          </input>
        </div>
        <div>
           email: <input placeholder="Email id" type="email"></input>
        </div>
        <div>
           password: <input type="password" placeholder="Password">
           </input>
        </div>
           <button type="submit">Add User</button>
       </table>
     </form>
    <script src="http://cdnjs.cloudflare.com/ajax/libs/jquery/1.8.3/
     jquery.min.js"></script>
    <script src="http://cdnjs.cloudflare.com/ajax/libs/knockout
       /2.2.0/knockout-min.js"></script>
    <link href="http://netdna.bootstrapcdn.com/twitter-
     bootstrap/2.3.2/css/bootstrap-combined.min.css" rel="stylesheet">
    <!-- <script src="http://ajax.aspnetcdn.com/ajax/jquery/jquery-
    1.9.0.js"></script> -->
    <script src="http://netdna.bootstrapcdn.com/twitter-
      bootstrap/2.3.2/js/bootstrap.min.js"></script>
  </body>
</html>
```

目前，HTML 页面仅显示空字段，如果你尝试输入数据并提交将什么也不会发生，因为还没有与后端服务完成数据绑定。

现在我们已经准备好创建 JavaScript，以用于对后端服务进行 REST 调用，并添加在 HTML 页面中输入的用户内容。

使用 Observable 和 AJAX

为了从 RESTful API 获取数据，我们会使用 AJAX。Observable 可以在数据更改时，自动在 `ViewModel` 定义的所有位置上更新。

使用 Observable，UI 和 `ViewModel` 动态通信将变得非常容易。

创建一个名为 `app.js` 的文件，其中已声明了 Observable。在 static 目录中，使用以下代码——如果目录不存在，请手动创建：

```
function User(data) {
  this.id = ko.observable(data.id);
  this.name = ko.observable(data.name);
  this.username = ko.observable(data.username);
  this.email = ko.observable(data.email);
  this.password = ko.observable(data.password);
}

function UserListViewModel() {
 var self = this;
 self.user_list = ko.observableArray([]);
 self.name = ko.observable();
 self.username = ko.observable();
 self.email = ko.observable();
 self.password = ko.observable();
 self.addUser = function() {
  self.save();
  self.name("");
  self.username("");
  self.email("");
  self.password("");
 };
self.save = function() {
  return $.ajax({
  url: '/api/v1/users',
  contentType: 'application/json',
  type: 'POST',
  data: JSON.stringify({
```

```
      'name': self.name(),
      'username': self.username(),
      'email': self.email(),
      'password': self.password()
   }),
   success: function(data) {
      alert("success")
           console.log("Pushing to users array");
           self.push(new User({ name: data.name, username:
           data.username, email: data.email, password:
            data.password }));
           return;
   },
   error: function() {
      return console.log("Failed");
   }
 });
   };
   }

 ko.applyBindings(new UserListViewModel());
```

上面的代码比较多，我们将按代码段来讲解。

当你在 HTML 页面上提交内容时，app.js 将收到一个请求，以下代码将处理该请求：

```
ko.applyBindings(new UserListViewModel());
```

其创建模型并将内容发送到下面的函数：

```
self.addUser = function() {
  self.save();
  self.name("");
  self.username("");
  self.email("");
  self.password("");
};
```

上面的 addUser 函数调用 self.save 方法并传递数据对象。save 函数对后台服务进行 AJAX RESTful 调用，并使用从 HTML 页面收集的数据执行 POST 操作。然后清除 HTML 页面的内容。

到这里还没完，我们前面提到双向数据绑定，因此我们还需要从 HTML 端发送数据，以便在数据库中做进一步处理。

在 s**cript** 标签中，添加下面一行，以定义.js 文件的路径：

```
<script src="{{ url_for('static', filename='app.js') }}"></script>
```

绑定数据到 adduser 模板

数据绑定功能对于将数据绑定到 UI 是非常有用的。如果不使用 Observable，UI 的属性仅会在首次加载的时候被处理。在这种情况下，UI 将无法根据底层数据的更新而自动更新。为了实现这一点，绑定必须引用 Observable 属性。

现在我们需要将数据与表单及其字段进行绑定，如下面的代码所示：

```
<form data-bind="submit: addUser">
 <div class="navbar">
   <div class="navbar-inner">
       <a class="brand" href="#">Tweet App Demo</a>
   </div>
 </div>
 <div id="main" class="container">
  <table class="table table-striped">
   Name: <input data-bind="value: name" placeholder="Full Name of
   user" type "text"/>
 </div>
 <div>
   Username: <input data-bind="value: username"
    placeholder="Username" type="username"></input>
 </div>
<div>
  email: <input data-bind="value: email" placeholder="Email id"
  type="email"></input>
</div>
```

```
<div>
   password: <input data-bind="value: password" type="password"
   placeholder="Password"></input>
</div>
   <button type="submit">Add User</button>
</table>
</form>
```

下面，我们准备通过模板添加用户。但是，如何来验证用户已被成功添加到数据库中了呢？一种方法是手动登录数据库查看。但是，由于我们开发的是 Web 应用程序，所以不如直接在网页上显示数据（存在数据库中）——即使是新添加的条目。

在 app.js 中添加如下代码，以读取数据库获得用户列表：

```
$.getJSON('/api/v1/users', function(userModels) {
  var t = $.map(userModels.user_list, function(item) {
    return new User(item);
  });
 self.user_list(t);
});
```

在 adduser.html 中增加显示用户列表的代码，如下所示：

```
<ul data-bind="foreach: user_list, visible: user_list().length > 0">
  <li>
    <p data-bind="text: name"></p>
    <p data-bind="text: username"></p>
    <p data-bind="text: email"></p>
    <p data-bind="text: password"></p>
  </li>
</ul>
```

至此，我们已经完成了为应用程序添加新用户的 Web 页面，如下图所示。

Tweet App Demo

Name: Full Name of user
Username: Username
email: Email id
password: Password
Add User

- Eric stromberg

  eric.strom

  eric.strom@google.com

  eric@123

用户发送推文

在开始编写这部分 Web 页面前，需要先为发送推文的方法创建路由，如下所示：

```
from flask import render_template

@app.route('/addtweets')
def addtweetjs():
  return render_template('addtweets.html')
```

创建了路由后，在 `addtweets.html` 中创建另一个表单，这样就可以向用户询问推文的相关信息，并帮助他们提交信息：

```
<html>
 <head>
  <title>Twitter Application</title>
 </head>
<body>
<form>
```

```
    <div class="navbar">
     <div class="navbar-inner">
         <a class="brand" href="#">Tweet App Demo</a>
     </div>
    </div>
    <div id="main" class="container">
     <table class="table table-striped">
       Username: <input placeholder="Username" type="username">
        </input>
    </div>
    <div>
      body: <textarea placeholder="Content of tweet" type="text">
      </textarea>
    </div>
    <div>
    </div>
     <button type="submit">Add Tweet</button>
    </table>

   </form>
    <script src="http://cdnjs.cloudflare.com/ajax/libs/jquery/1.8.3/
     jquery.min.js"></script>
    <script src="http://cdnjs.cloudflare.com/ajax/libs/
     knockout/2.2.0/knockout-min.js"></script>
     <link href="http://netdna.bootstrapcdn.com/twitter-
       bootstrap/2.3.2/css/bootstrap-combined.min.css" rel="stylesheet">
     <!-- <script src="http://ajax.aspnetcdn.com/ajax/jquery/jquery-
       1.9.0.js"></script> -->
     <script src="http://netdna.bootstrapcdn.com/twitter-
       bootstrap/2.3.2/js/bootstrap.min.js"></script>
    </body>
   </html>
```

请注意，此表单还没有与 RESTful 服务进行数据绑定。

在推文模板上使用 Observable 和 AJAX

我们要开发一个可以对后端服务进行 REST 调用的 JavaScript，并添加从 HTML 页面提供的推文内容。

在 static 目录下创建一个名为 `tweet.js` 的文件，其中包括如下代码：

```
function Tweet(data) {
  this.id = ko.observable(data.id);
  this.username = ko.observable(data.tweetedby);
  this.body = ko.observable(data.body);
  this.timestamp = ko.observable(data.timestamp);
}

function TweetListViewModel() {
  var self = this;
  self.tweets_list = ko.observableArray([]);
  self.username = ko.observable();
  self.body = ko.observable();

  self.addTweet = function() {
  self.save();
  self.username("");
  self.body("");
   };

  $.getJSON('/api/v2/tweets', function(tweetModels) {
  var t = $.map(tweetModels.tweets_list, function(item) {
   return new Tweet(item);
  });
  self.tweets_list(t);
  });
  self.save = function() {
   return $.ajax({
   url: '/api/v2/tweets',
   contentType: 'application/json',
   type: 'POST',
```

```
      data: JSON.stringify({
         'username': self.username(),
         'body': self.body(),
      }),
      success: function(data) {
         alert("success")
              console.log("Pushing to users array");
              self.push(new Tweet({
              username: data.username,
              body: data.body
              }));
              return;
      },
      error: function() {
         return console.log("Failed");
      }
     });
      };
    }

ko.applyBindings(new TweetListViewModel());
```

下面分别来讲解代码的每个部分。

当你在 HTML 页面上提交内容时，将会向 `tweet.js` 发送请求，以下代码将处理该请求：

```
ko.applyBindings(new TweetListViewModel());
```

上述代码片段创建了模型并将内容发送到如下函数：

```
self.addTweet = function() {
  self.save();
  self.username("");
  self.body("");
  };
```

`addTweet` 函数使用传入的数据对象调用 `self.save` 函数。save 函数对后端服务进行 AJAX RESTful 调用，并使用从 HTML 页面收集的数据执行 POST 操作。然后清除 HTML

页面的内容。

为了让页面上显示的数据与后端保持同步，需要添加如下代码：

```
function Tweet(data) {
  this.id = ko.observable(data.id);
  this.username = ko.observable(data.tweetedby);
  this.body = ko.observable(data.body);
  this.timestamp = ko.observable(data.timestamp);
}
```

至此，工作还没结束，还要进行双向数据绑定，为此，我们还需要从 HTML 端发送数据，以便其在数据库中被进一步处理。

在 script 标签中，添加下面一行，来指定.js 文件的路径：

```
<script src="{{ url_for('static', filename='tweet.js') }}"></script>
```

绑定数据到 addtweet 模版

现在需要将数据与表单及其字段进行绑定，如下面的代码所示：

```
<form data-bind="submit: addTweet">
  <div class="navbar">
    <div class="navbar-inner">
       <a class="brand" href="#">Tweet App Demo</a>
    </div>
   </div>
   <div id="main" class="container">

    <table class="table table-striped">
      Username: <input data-bind="value: username"
      placeholder="Username" type="username"></input>
   </div>
   <div>
      body: <textarea data-bind="value: body" placeholder="Content
      of tweet" type="text"></textarea>
   </div>
   <div>
```

```
   </div>
   <button type="submit">Add Tweet</button>
   </table>
</form>
```

现在可以通过模板添加推文了。就像在前面验证用户一样验证推文。

在 `tweet.js` 文件中添加如下代码读取数据库并获取推文列表：

```
$.getJSON('/api/v2/tweets', function(tweetModels) {
  var t = $.map(tweetModels.tweets_list, function(item) {
  return new Tweet(item);
 });
  self.tweets_list(t);
});
```

修改 `addtweets.html` 页面以显示推文列表，如下：

```
<ul data-bind="foreach: tweets_list, visible: tweets_list().length > 0">
<li>
  <p data-bind="text: username"></p>
  <p data-bind="text: body"></p>
  <p data-bind="text: timestamp"></p>

 </li>
</ul>
```

进行测试，将会看到如下图所示的页面。

以同样的方式，你可以通过在 Web 应用程序中删除用户来扩展此用例，也可以更新后台服务中的用户信息。

此外，要了解更多关于 knockout.js 库的信息，请参阅 `http://knockoutjs.com/examples/helloWorld.html` 上的实时示例，这将有助于你更好地了解该库，并在应用程序中使用。

我们创建了这些网页，确保微服务能够正常工作，并带你了解了如何开发 Web 应用程序。开发者可以根据自己的用例创建 Web 应用程序。

CORS——跨源资源共享

CORS 有助于在 API 服务器和客户端之间保持 API 请求的数据完整性。

使用 CORS 的理念是，服务器和客户端之间应该有足够的信息便于它们相互认证，并使用 HTTP header 通过安全通道传输数据。

当客户端进行 API 调用时，其通常是一个 GET 或 POST 请求，其中 body 通常是 text/plain，header 为 **Origin**——包括请求页面的协议、域名和端口。服务器确认请求并将响应与 `Access-Control-Allow-Origin` header 一起发送到同一个 Origin，确保响应被正确的 Origin 接收到。

通过这种方式在 Origin 之间共享资源。

几乎所有的浏览器都支持 CORS，包括 IE 8+、Firefox 3.5+和 Chrome。

目前我们的 Web 应用程序还没有启用 CORS，我们来启用一下吧。

首先使用下面的命令在 Flask 中安装 CORS 模块：

```
$pip install flask-cors
```

这个包中包含了一个 Flask 扩展，它默认启用了支持所有 Origin 和方法路由的 CORS。在 `app.py` 中引入这个包：

```
from flask_cors import CORS, cross_origin
```

要启用 CORS，需要在代码中添加下面一行：

```
CORS(app)
```

现在我们已经为 Flask 应用中的所有资源启用了 CORS。

如果要在特定资源上启用 CORS，请在特定资源中添加以下代码：

```
cors = CORS(app, resources={r"/api/*": {"origins": "*"}})
```

到目前为止，我们还没有设置域名，一直是在 localhost 上运行。你可以添加一个自定义域名来测试 CORS，如下：

```
127.0.0.1       <your-domain-name>
```

现在可以尝试访问<your-domain-name>，应该可以正常工作了，可以访问该资源。

Session 管理

session（会话）是与单个用户相关联的请求和响应事务的顺序。Session 通常用于服务器端用户认证和通过网页跟踪用户活动。

每个客户端的 session 都拥有一个 session ID。session 通常存储在 cookie 的顶部，服务器以密码方式签名——Flask 应用程序使用密钥解密获取 session 的有效时间。

目前，我们还没有设置身份验证——将在第 8 章中定义。所以，现在我们在页面中获取用户名，并确保使用 session 标识用户来创建会话。

创建一个名为 main.html 的 Web 页面，如果需要设置 session，会有一个 URL 来创建 session，并且路由对后端服务执行操作。如果 session 已经存在，可以清除该 session，如以下代码所示：

```
<html>
  <head>
    <title>Twitter App Demo</title>
    <link rel=stylesheet type=text/css href="{{ url_for('static',
    filename='style.css') }}">
</head>
<body>
    <div id="container">
```

```
        <div class="title">
          <h1></h1>
        </div>
        <div id="content">
          {% if session['name'] %}
          Your name seems to be <strong>{{session['name']}}</strong>.
         <br/>
          {% else %}
          Please set username by clicking it <a href="{{
          url_for('addname') }}">here</a>.<br/>
          {% endif %}
         Visit <a href="{{ url_for('adduser') }}">this for adding new
         application user </a> or <a href="{{ url_for('addtweetjs') }}">this
         to add new tweets</a> page to interact with RESTFUL API.

         <br /><br />
         <strong><a href="{{ url_for('clearsession') }}">Clear
         session</a></strong>
         </div>
         </div>
      </div>
    </body>
</html>
```

现在页面上的一些 URL，例如 clearsession 和 addname 还不能工作，因为我们还没为它们添加路由。

而且我们也没为 main.html 页面设置路由。在 app.py 中添加路由，如下所示：

```
@app.route('/')

def main():
  return render_template('main.html')
```

然后在 app.py 中为 addname 添加路由，如下所示：

```
@app.route('/addname')
```

```
def addname():
if request.args.get('yourname'):
 session['name'] = request.args.get('yourname')
 # And then redirect the user to the main page
   return redirect(url_for('main'))

 else:
   return render_template('addname.html', session=session)
```

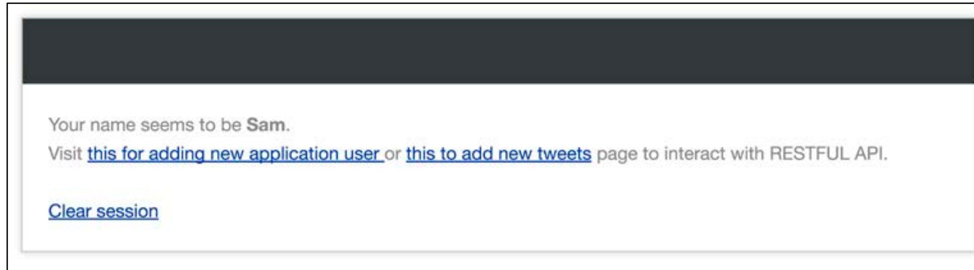
由以上路由代码可以看到，调用的 addname.html 文件还没创建。

使用下面的代码创建 addname 模板：

```
 <html>
  <head>
    <title>Twitter App Demo</title>
    <link rel=stylesheet type=text/css href="{{ url_for('static',
    filename='style.css') }}">
  </head>
<body>
    <div id="container">
        <div class="title">
            <h1>Enter your name</h1>
        </div>
    <div id="content">
      <form method="get" action="{{ url_for('addname') }}">
        <label for="yourname">Please enter your name:</label>
         <input type="text" name="yourname" /><br />
         <input type="submit" />
      </form>
    </div>
    <div class="title">
           <h1></h1>
    </div>
    <code><pre>
    </pre></code>
   </div>
  </div>
```

```
 </body>
/html>
```

使用上述代码设置好了 sesson 后，你将看到下图所示的页面。



现在，如果我们需要清除 session，该怎么办呢？由于我们已经从主页调用了 `clearsession` 函数，所以需要在 `app.py` 中创建一个路由，进一步调用 sesson 的内置 `Clear` 函数，如下所示：

```
@app.route('/clear')

def clearsession():
  # Clear the session
  session.clear()
  # Redirect the user to the main page
  return redirect(url_for('main'))
```

这段代码显示了，如何设置 session，维护用户信息，并根据要求清除 session。

Cookies

cookie 类似于 session，只不过它是以文本文件的形式存在于客户端计算机上，而不是在服务器端维护。

cookie 的主要目的是跟踪客户的使用情况，并根据用户活动，来改善体验。

cookie 属性存储在响应对象中，该对象是具有 cookie、变量及其相应值的键/值对的集合。

我们可以使用 `set_cookie()`函数为响应对象设置 cookie：

```
@app.route('/set_cookie')
def cookie_insertion():
  redirect_to_main = redirect('/')
  response = current_app.make_response(redirect_to_main )
  response.set_cookie('cookie_name',value='values')
  return response
```

同样读取 cookie 也很简单，使用 `get()`函数获取已存在的 cookie，如下所示：

```
import flask
cookie = flask.request.cookies.get('my_cookie')
```

如果 cookie 已存在，将会获取指定的 cookie，如果不存在将会返回 `None`。

本章小结

在本章中，我们学习了如何在 Web 应用程序中使用 JavaScript 库（如 knockout.js）并与微服务集成。了解了 MVVM 模式如何帮助我们构建 Web 应用程序。还学习了用户管理的相关概念，如 cookie 和 session，以及如何使用它们。

在下一章中，我们将会把数据库从 SQLite 迁移到 NoSQL（如 MongoDB），这将使数据库更加强大和安全。

4

与数据服务交互

在上一章中，我们使用 JavaScript 和 HTML 构建了应用程序，并将其与 AJAX RESTful API 集成在一起。学习了如何在服务器上为客户端和 session 设置 cookie，以便为用户提供更好的体验。本章，我们将重点使用 NoSQL 数据库（如 MongoDB）而不是前面的 SQLite 数据库或 MySQL 数据库，增强后端数据库功能，并将其与应用程序集成。

本章主要包含如下内容：

- 安装 MongoDB
- 在应用程序中集成 MongoDB

MongoDB 有什么优势，为什么要使用它

在安装 MongoDB 之前，我们先了解一下为什么选择 MongoDB。

MongoDB 相对于 RDBMS（关系型数据库管理系统）有如下几个优势。

- **灵活的模式（schema）：**MongoDB 是一种文档数据库，其中一个集合包含多个文档。在插入数据之前不需要定义文档的模式，MongoDB 会根据插入文档中的数据来定义文档的模式；而在 RDBMS 中，需要在将数据插入之前就定义好表的模式。
- **复杂性低：**MongoDB 中没有 RDBMS（例如 MySQL）中那么复杂的 join。
- **易扩展：**相较于 RDBMS 更容易扩展。

- **访问速度快：**与 RDBMS（如 MySQL 数据库）相比，MongoDB 中的数据检索速度更快。
- **动态查询：**MongoDB 支持对文档进行动态查询，作为一种基于文档的查询语言，它比其他 RDBMS（例如 MySQL）更有优势。

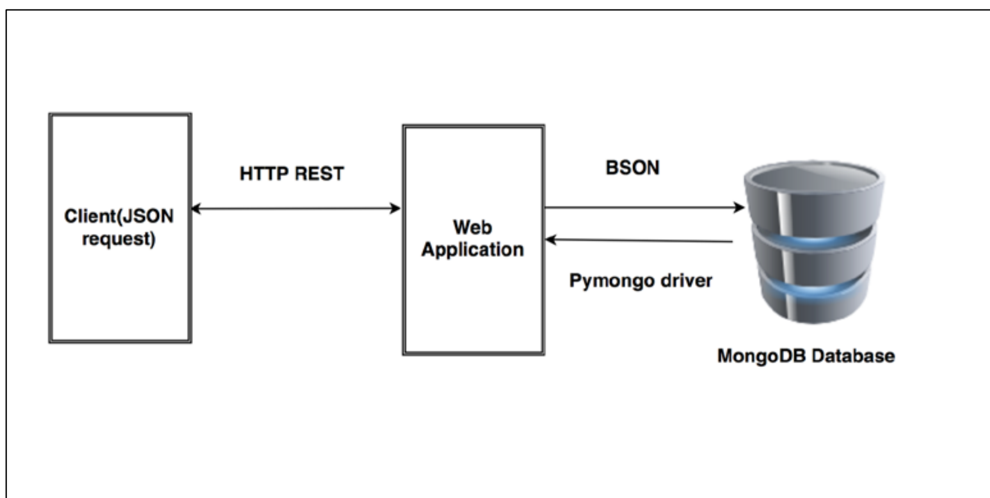
下面列出了几个使用 MongoDB 数据库的原因：

- MongoDB 使用 JSON 格式文档存储数据，这使得其很容易与其应用程序集成。
- 可以在任何文件和属性上设置索引。
- MongoDB 自动分片，这样可以轻松管理并提高运行速度。
- 在集群模式时可以实现副本控制和高可用。

MongoDB 有如下几类用途：

- 大数据
- 用户数据管理
- 内容交付和管理

下图是 MongoDB 与 Web 应用程序集成的架构图。



MongoDB 中的术语

我们先来熟悉下 MongoDB 中的一些术语。

- **数据库**：与 RDBMS（关系型数据库管理系统）中的数据库概念类似，但在 MongoDB 中，数据库是集合的物理容器，而不是基于表的。 MongoDB 中可以有多个数据库。
- **集合**：集合是各种模式的文档的组合。集合对文档的架构没有影响，相当于 RDBMS 中的表。
- **文档**：这与 RDBMS 中的元组/行类似。它是一组键值对，具有动态模式，每个文档在单个集合中可以具有相同或不同的模式。每个文档也可能有不同的字段。

下面的代码可以帮助你理解集合的概念：

```
{
   _id: ObjectId(58 ccdd1a19b08311417b14ee),
   body: 'New blog post,Launch your app with the AWS Startup Kit!#
   AWS ',
   timestamp: "2017-03-11T06:39:40Z",
   id: 18,
   tweetedby: "eric.strom"
}
```

MongoDB 以二进制编码格式表示 JSON 文档，称为 BSON。

安装 MongoDB

由于我们当前使用的是 Ubuntu 工作站，因此按照下列步骤在 Ubuntu 上安装 MongoDB。

使用 Ubuntu 软件包管理工具（如 `apt`）和 GPG 密钥验证经签名的软件包来安装 MongoDB。

执行以下命令导入 GPG 密钥：

```
$ sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv
EA312927
```

将下面的 MongoDB 源加入操作系统中：

```
$ echo "deb http://repo.mongodb.org/apt/ubuntu trusty/mongodb-org/3.2
multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-3.2.list
```

更新 Ubuntu 软件源：

```
$ sudo apt-get update
```

安装最新版本的 MongoDB 发行版：

```
$ sudo apt-get install -y mongodb-org
```

安装完成后，MongoDB 将会启用 27017 端口。检查服务状态：

```
$ sudo service mongodb status
```

如果服务没有运行，则执行下列命令启动服务：

```
$ sudo service mongodb start
```

这样就成功地在本地安装了 MongoDB，但是目前我们只安装了一个 MongoDB 实例，这对我们来说足够了。如果你想创建 MongoDB 集群，请参考下面链接中的介绍：

https://docs.mongodb.com/manual/tutorial/deploy-shard-cluster/

下面我们就在该 MongoDB 集群上创建一个数据库！

初始化 MongoDB 数据库

我们在前面章节中使用 SQLite3 数据库时需要手动创建数据库和定义表结构。由于 MongoDB 是无模式的，因此可以直接向其中添加新文档，MongoDB 会自动创建集合。在这种情况下，我们使用 Python 来初始化数据库。

在向 MongoDB 中添加文档之前，需要先安装 MongoDB 的数据库驱动 pymongo。

在 requirements.txt 文件中增加 pymongo 的配置，然后使用 pip 命令安装软件包：

```
$ echo "pymongo==3.4.0" >> requirements.txt
$ pip install -r requirements.txt
```

然后在 app.py 中引入：

```
from pymongo import MongoClient
```

在 app.py 中定义一个函数来创建 MongoDB 数据库连接，使用初始数据文档来初始

化数据库：

```
connection = MongoClient("mongodb://localhost:27017/")
def create_mongodatabase():
try:
   dbnames = connection.database_names()
   if 'cloud_native' not in dbnames:
       db = connection.cloud_native.users
       db_tweets = connection.cloud_native.tweets
       db_api = connection.cloud_native.apirelease
       db.insert({
       "email": "eric.strom@google.com",
       "id": 33,
       "name": "Eric stromberg",
       "password": "eric@123",
       "username": "eric.strom"
       })

       db_tweets.insert({
       "body": "New blog post,Launch your app with the AWS Startup Kit! #AWS",
       "id": 18,
       "timestamp": "2017-03-11T06:39:40Z",
       "tweetedby": "eric.strom"
       })

       db_api.insert( {
       "buildtime": "2017-01-01 10:00:00",
       "links": "/api/v1/users",
       "methods": "get, post, put, delete",
       "version": "v1"
       })

       db_api.insert( {
         "buildtime": "2017-02-11 10:00:00",
         "links": "api/v2/tweets",
         "methods": "get, post",
         "version": "2017-01-10 10:00:00"
```

```
        })
        print ("Database Initialize completed!")
    else:
        print ("Database already Initialized!")
    except:
        print ("Database creation failed!!")
```

建议使用集合中的某些文档来初始化资源集合，以便在开始测试 API 时能够获取一些响应数据，也可以在不初始化集合的情况下继续下面的步骤。

在启动应用程序之前应该调用前面的函数。main 函数如下所示：

```
if __name__ == '__main__':
  create_mongodatabase()
  app.run(host='0.0.0.0', port=5000, debug=True)
```

在微服务中集成 MongoDB

初始化了 MongoDB 数据库后,我们重写微服务,将原来的 SQLite 3 替换为 MongoDB。

之前我们使用 curl 命令获取 API 的响应，现在我们使用一款名为 **POSTMAN**（https://www.getpostman.com）的工具来获取，使用它可以更加快速地构建、测试和编写 API。



了解关于 POSTMAN 的更多信息请访问该链接：https://www.getpostman.com/docs/

POSTMAN 同时支持 Chrome 和 Firefox，可以使用插件的方式安装。

首先，修改 api_version info API，将原来从 SQLite3 中获取信息的方式改成从 MongoDB 中获取：

```
@app.route("/api/v1/info")
def home_index():
 api_list=[]
  db = connection.cloud_native.apirelease
  for row in db.find():
```

```
    api_list.append(str(row))
  return jsonify({'api_version': api_list}), 200
```

使用 POSTMAN 测试，将看到如下图所示的输出。



成功！下面我们将更新其他微服务资源。

处理 user 资源

下面将更新 `app.py` 中的所有用户资源 API 方法。

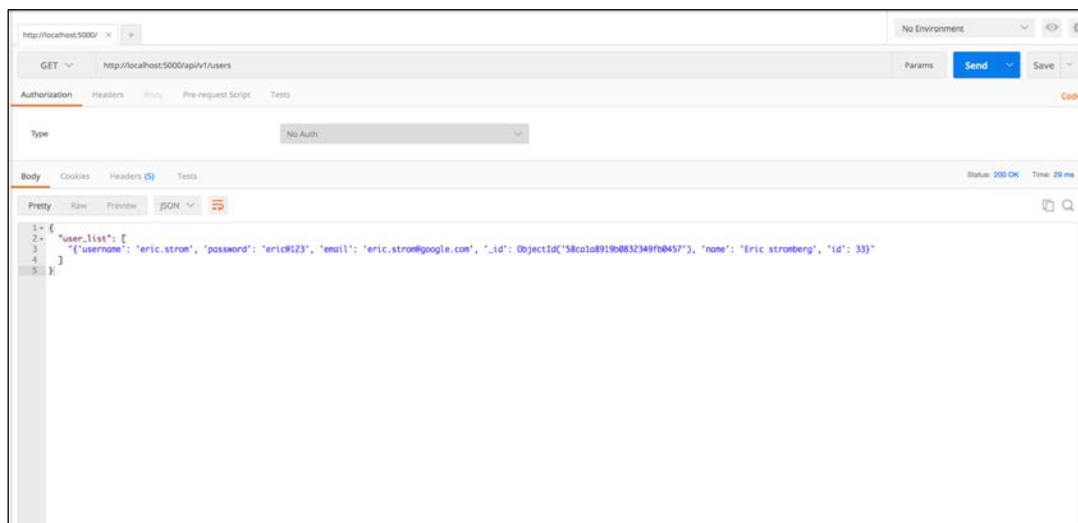
GET api/v1/users

GET API 函数可以获取用户的完整列表。

重写 `list_users()`函数以从 MongoDB 数据库中获取全部用户的列表。

```
def list_users():
  api_list=[]
  db = connection.cloud_native.users
  for row in db.find():
    api_list.append(str(row))
  return jsonify({'user_list': api_list})
```

使用 POSTMAN 测试，看看 API 的响应是否跟我们期望的一样，如下图所示。



因为当前 MongoDB 数据库下的用户集合中只有一个文档，所以该图中只显示一条用户信息。

GET api/v1/users/[user_id]

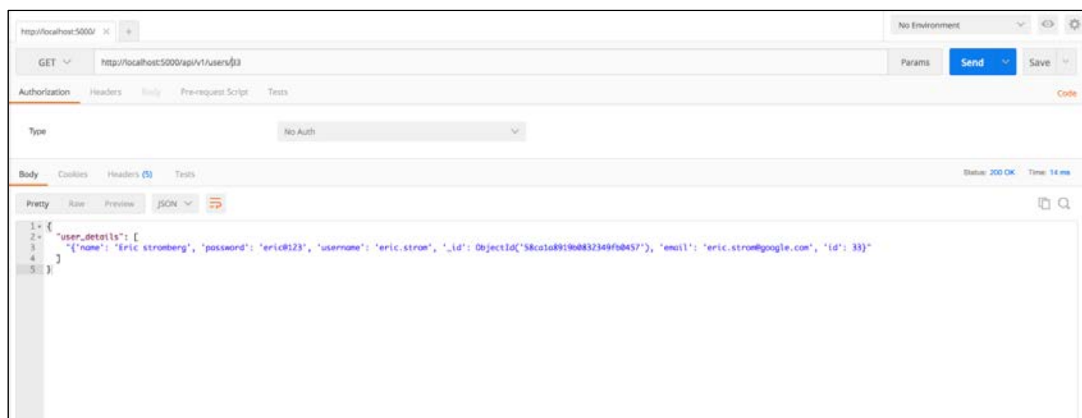
该 API 获取指定用户的详细信息。

修改 `list_user(user_id)`函数以从 MongoDB 中获取指定的用户信息：

```
def list_user(user_id):
 api_list=[]
 db = connection.cloud_native.users
 for i in db.find({'id':user_id}):
   api_list.append(str(i))

 if api_list == []:
   abort(404)
 return jsonify({'user_details':api_list}
```

使用 POSTMAN 测试，看看 API 的响应是否跟我们期望的一样，如下图所示。



测试用户不存在的情况，结果应该如下图所示。



POST api/v1/users

该 API 向用户列表中添加新用户。

下面，我们将重写 `add_user(new_user)`函数来跟 MongoDB 交互，向用户集合中添加一个新用户：

```
def add_user(new_user):
 api_list=[]
 print (new_user)
 db = connection.cloud_native.users
 user = db.find({'$or':[{"username":new_user['username']},
```

```
{"email":new_user['email']}]})
 for i in user:
   print (str(i))
   api_list.append(str(i))

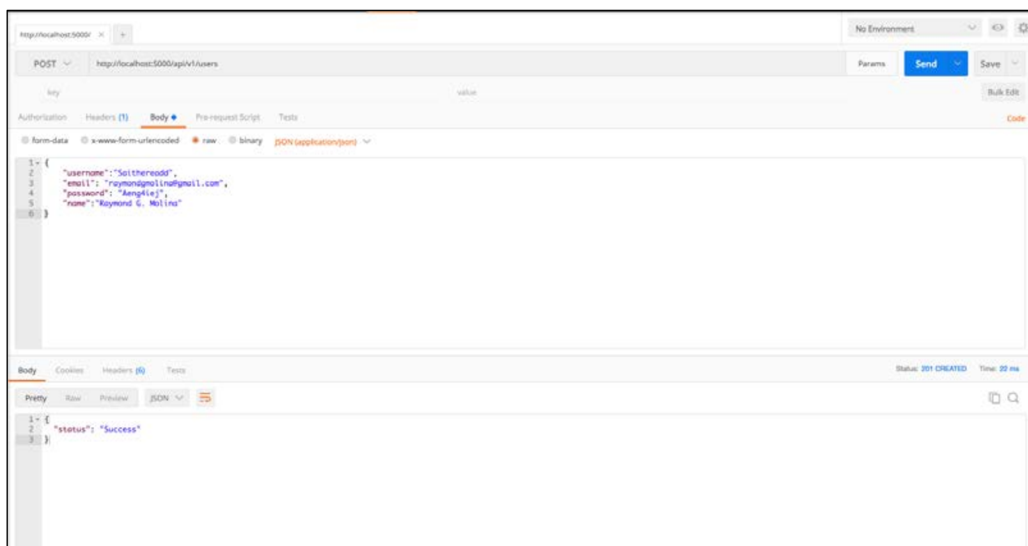
 if api_list == []:
   db.insert(new_user)
   return "Success"
 else :
   abort(409)
```

修改函数后，还需要做一件事情——之前使用 SQLite3 时 ID 是自动生成的，现在需要使用随机数模块来生成 ID：

```
def create_user():
 if not request.json or not 'username' in request.json or not
'email' in request.json or not 'password' in request.json:
   abort(400)
 user = {
   'username': request.json['username'],
   'email': request.json['email'],
   'name': request.json.get('name',""),
   'password': request.json['password'],
   'id': random.randint(1,1000)
}
```

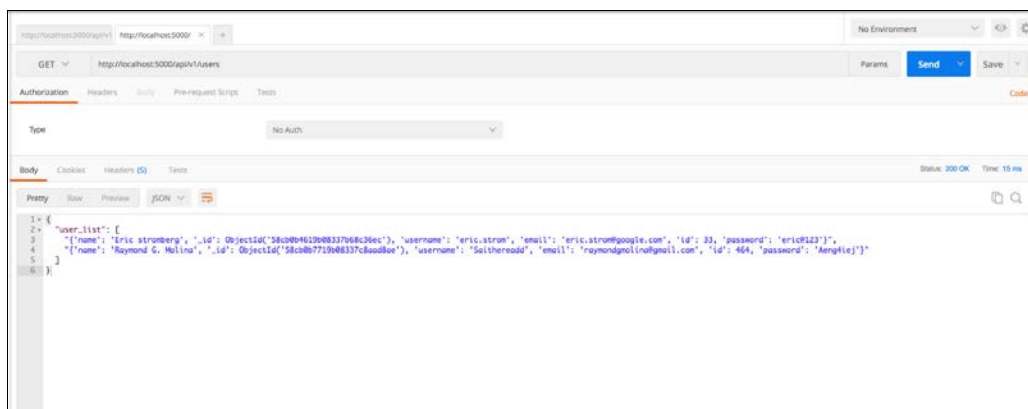
向用户列表中添加一条记录，看看是否有效。

如下图所示，使用 POSTMAN 向 MongoDB 中添加了一条记录。



再来验证下是否成功更新了 MongoDB 集合中的属性。

如下图所示，成功增加了一条记录。



PUT api/v1/users/[user_id]

该 API 用来更新 MongoDB 用户集合中指定用户的属性。

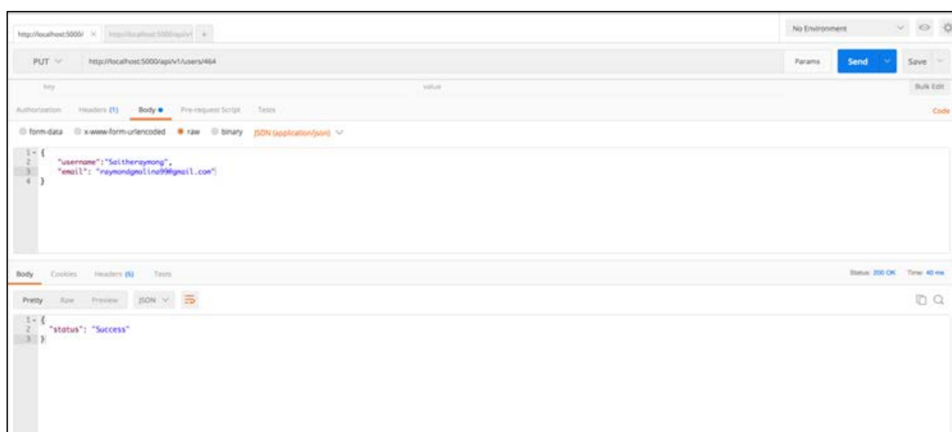
重写 `upd_user(user)`方法以更新 MongoDB 用户集合中指定用户的文档：

```
def upd_user(user):
```

```
api_list=[]
print (user)
db_user = connection.cloud_native.users
users = db_user.find_one({"id":user['id']})
for i in users:
  api_list.append(str(i))
 if api_list == []:
  abort(409)
 else:
  db_user.update({'id':user['id']},{'$set': user}, upsert=False )
  return "Success"
```

更新该方法后，使用 POSTMAN 来测试。

如下图所示，使用 POSTMAN 测试更新后的 API 请求的响应。



验证用户文档，查看字段是否被修改，如下图所示。



DELETE api/v1/users

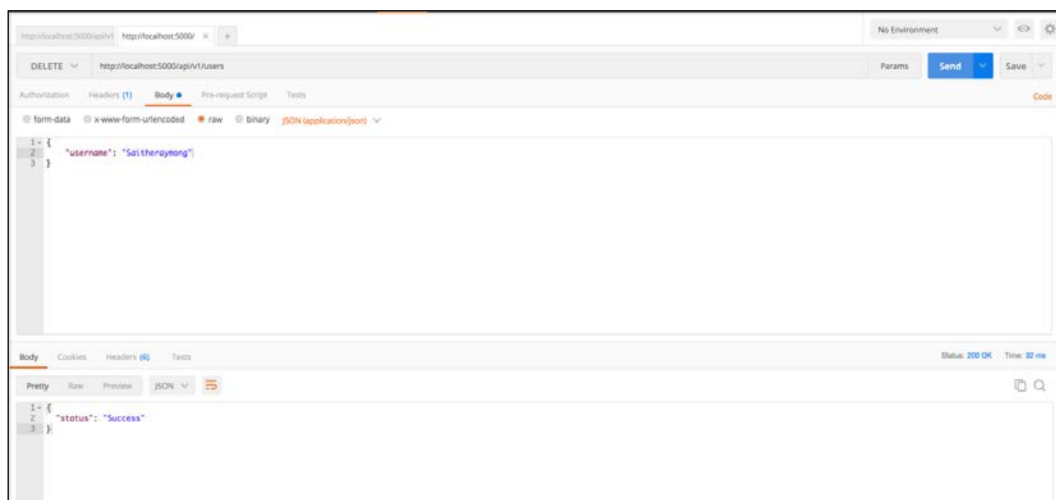
该 API 从用户列表中删除指定用户。

修改 `del_user(del_user)`方法将指定用户从 MongoDB 用户集合中删除：

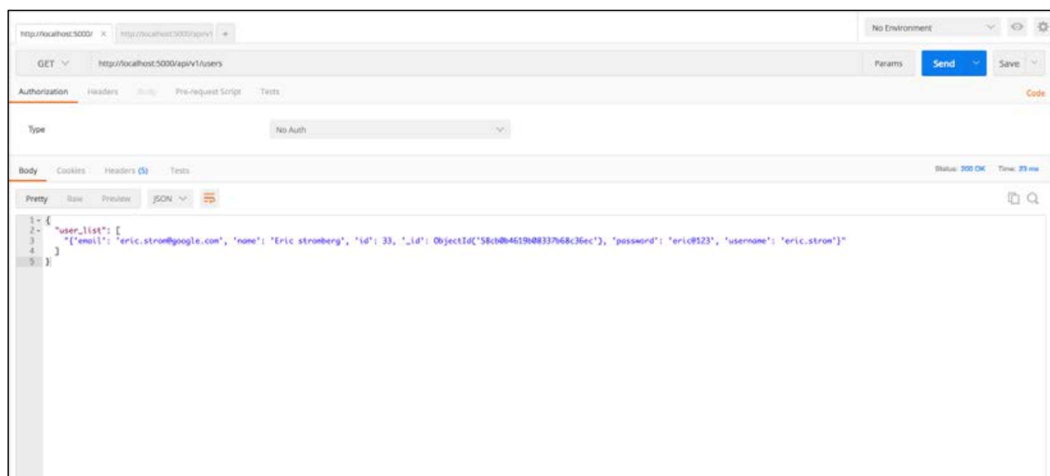
```
def del_user(del_user):
  db = connection.cloud_native.users
 api_list = []
 for i in db.find({'username':del_user}):
    api_list.append(str(i))

 if api_list == []:
    abort(404)
else:
  db.remove({"username":del_user})
  return "Success"
```

使用 POSTMAN 测试，看看 API 的响应是否跟我们期望的一样，如下图所示。



我们刚刚删除了一条用户记录，现在来看看用户列表是否已更新，如下图所示。



至此我们修改了所有用户资源的 RESTful API URL 并验证通过。

处理推文资源

现在用户资源的 API 已经成功适配 MongoDB 数据库服务，接下来我们再来对推文资源做同样的修改。

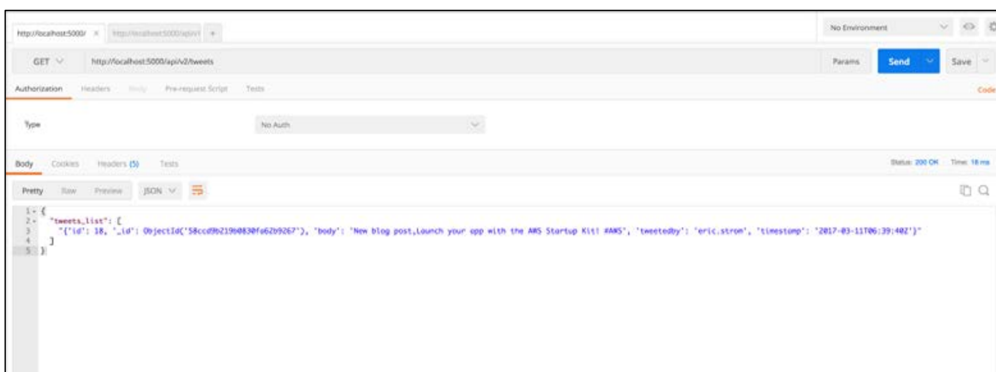
GET api/v2/tweets

该 API 用于获取所有用户的所有推文列表。

修改 `list_tweets()`方法以从 MongoDB 中获取所有用户的所有推文：

```
def list_tweets():
    api_list=[]
    db = connection.cloud_native.tweet
    for row in db.find():
    api_list.append(str(row))
    return jsonify({'tweets_list': api_list})
```

修改完代码后使用 POSTMAN 测试。如下图所示，使用 POSTMAN 请求获取所有推文的 API。



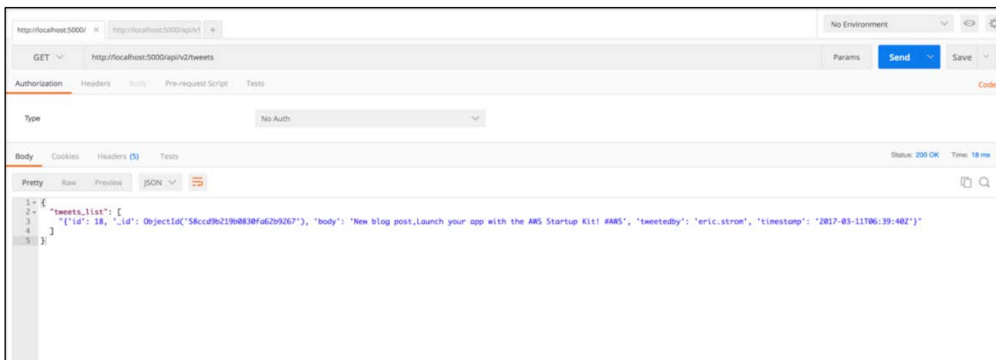
GET api/v2/tweets/[user_id]

该 API 用于获取指定用户的推文。

修改 `list_tweet(user_id)` 函数以从推文集合中获取指定用户的推文。

```
def list_tweet(user_id):
 db = connection.cloud_native.tweets
 api_list=[]
 tweet = db.find({'id':user_id})
 for i in tweet:
   api_list.append(str(i))
if api_list == []:
   abort(404)
return jsonify({'tweet': api_list})
```

验证该 API 是否有效，如下图所示。



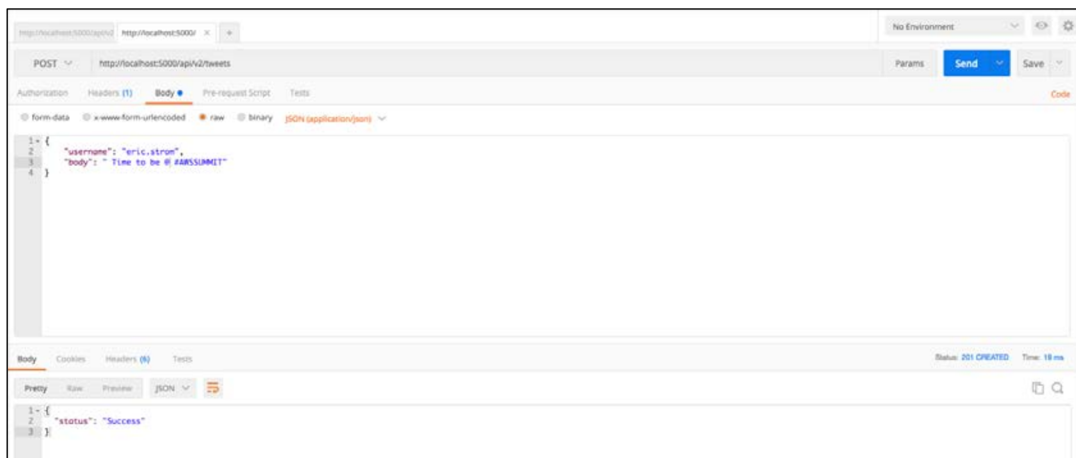
POST api/v2/tweets

该 API 向已知用户添加推文。

修改 add_tweet(new_tweet)方法，与用户交互，并向 MongoDB 中的推文集合中添加一条新推文：

```
def add_tweet(new_tweet):
  api_list=[]
  print (new_tweet)
  db_user = connection.cloud_native.users
  db_tweet = connection.cloud_native.tweets
  user = db_user.find({"username":new_tweet['tweetedby']})
  for i in user:
    api_list.append(str(i))
    if api_list == []:
     abort(404)
    else:
      db_tweet.insert(new_tweet)
      return "Success"
```

修改完后，使用 POSTMAN 测试。使用 POSTMAN 执行 POST 请求添加新推文后的成功状态如下图所示。



验证新添加的推文是否被成功添加到了推文列表中，如下图所示。



本章小结

在本章中，我们把基于文件的数据库服务（SQLite）迁移到了基于 NoSQL 的数据库服务（MongoDB）。学习了如何将 MongoDB 与 RESTful API 集成以保存数据，并根据客户端的请求进行响应。下一章的内容更有趣，因为我们将使用 React 构建前端 Web 视图。

5

使用 React 构建 Web 视图

到目前为止，我们一直在构建微服务，让后台服务更加灵活和高效。此外，还一直在尝试不同的数据库服务，以提高数据存储和检索的性能，这至关重要。

在本章中，我们将重点介绍如何使用 React 构建前端页面，并将这些页面与后端集成起来形成一个完整的应用程序。

本章包括的主要内容有：

- 配置 React 环境
- 创建用户验证面板
- 将 React 与后端 API 集成

理解 React

简单来说，React 是应用程序的 UI 层。它是一个用来构建快速响应用户请求的 JavaScript 库。React 可以帮助你创建 awesome webViews foreach 状态的应用程序，因此我们选择使用 React 构建 Web 视图。首先，我们来了解下 React 的概念和要点。

- **组件**：所有的 HTML 和 JavaScript 的集合都被称为组件。React 提供了一些钩子，它们可以使用 JavaScript 来创建 HTML 页面。React 作为一个控制器为应用程序的每个状态呈现不同的网页。

- **React 中静态版本的 Props**：通常，在 HTML 中，需要使用大量的重复代码来显示前端的所有数据。React 可以帮助你解决这个问题。Props 保持数据的状态，并将值从父节点传递给子节点。
- **识别最小状态**：要正确构建应用程序，首先需要考虑应用程序的最小可变状态集。就像我们的例子，需要在应用程序的不同状态下始终保持用户状态可用。
- **识别活动状态**：React 中的组件层次结构是基于单向数据流的。我们需要了解基于该状态呈现内容的每个组件。此外，还需要了解各个状态如何随着组件层次的变化而变化。
- **React-DOM**：react-dom 是 React 和 DOM 的组合。React 中包含 Web 和移动应用程序的功能合集。react-dom 包含 Web 应用程序的功能。

配置 React 环境

我们需要安装一系列 `node.js` 库来初始化 React 环境。

安装 node

安装 React 之前需要先安装 `node.js`。

在 Linux（Debian 系列）操作系统上，安装步骤十分简单。

首先，执行下面的命令添加 node.js 的官方 PPA：

```
$ sudo apt-get install python-software-properties
$ curl -sL https://deb.nodesource.com/setup_7.x | sudo -E bash -
```

然后执行下面的命令安装 `node.js`：

```
$ apt-get install nodejs
```

现在检查 ndoe 和 npm 的版本：

```
$ npm –v
  4.1.2
$ node –v
  V7.7.4
```

虽然我们安装了 npm4.1.2 及 node V7.7.4 版本，但你也可以使用 node v7.x 和 npm v4.x 版本。

创建 package.json

此文件是应用程序的元数据配置，其中包含需要为应用程序安装的完整库/依赖关系。该文件还有个好处，就是可以使你的构建可重复，也就是说方便你与其他开发人员共享。创建自定义的 package.json 有很多种方式。

下面是 packages.json 文件需要提供的最简配置：

```
Name：小写
version：格式为x.x.x
例如：
{
   "name": "my-twitter-package",
   "version": "1.0.0"
}
```

使用下面的命令创建 package.json 模板：

```
$ npm init           # in your workspace
```

该命令会询问你包的名称、版本、描述、作者、证书等信息，输入这些信息后，会生成 package.json 文件。

如果你不想输入这些信息，则可以指定--yes 或者-y 参数来使用默认值。

```
$npm init --yes
```

对于我们的应用程序，生成的 package.json 文件如下所示：

```
{
   "name": "twitter",
   "version": "1.0.0",
   "description": "Twitter App",
   "main": "index.js",
   "dependencies": {
     "babel-loader": "^6.4.1",
```

```
        "fbjs": "^0.8.11",
        "object-assign": "^4.1.1",
        "react": "^15.4.2",
        "react-dev": "0.0.1",
        "react-dom": "^0.14.7",
        "requirejs": "^2.3.3"
       },
      "devDependencies": {
       "babel-core": "^6.4.5",
       "babel-loader": "^6.2.1",
       "babel-preset-es2015": "^6.3.13",
       "babel-preset-react": "^6.3.13",
       "webpack": "^1.12.12"
      },
    "scripts": {
      "test": "echo \"Error: no test specified\" && exit 1"
     },
    "author": "Manish Sethi",
    "license": "ISC"
   }
```

生成了 package.json 文件后，使用下面的命令安装所有依赖：

```
$ npm install
```

确认执行了上面的命令，并且 package.json 文件存在于当前目录下。

使用 React 构建 webViews

首先，创建 React 调用的 home 视图。在模板目录下创建 index.html 文件，其中包含如下内容：

```
<!DOCTYPE html>
<html>
 <head lang="en">
  <meta charset="UTF-8">
  <title>Flask react</title>
```

```
 </head>
<body>
    <div class="container">
    <h1></h1>
    <br>
    <div id="react"></div>

 </div>

<!-- scripts -->
 <script src="https://code.jquery.com/jquery-2.1.1.min.js"></script>
 <script src="https://cdnjs.cloudflare.com/ajax/libs/
   react/15.1.0/react.min.js"></script>
 <script src="https://npmcdn.com/react-
   router@2.8.1/umd/ReactRouter.min.js"></script>
 <script src="https://cdnjs.cloudflare.com/ajax/libs/
   react/15.1.0/react-dom.min.js"></script>
 <script src="http://cdnjs.cloudflare.com/ajax/libs/
   react/0.13.3/JSXTransformer.js"></script>



 </body>
</html>
```

在该 HTML 页面中,我们定义了一个 `id="react"`的 `div`,我们将基于该 ID 调用 React 主函数执行相应的操作。

创建 `main.js`，其中发送响应的代码如下：

```
import Tweet from "./components/Tweet";
class Main extends React.Component {
render() {
  return (
  <div>
    <h1> Welcome to cloud - native - app! </h1>
  </div>
  );
 }
```

```
}



let documentReady = () => {
 ReactDOM.render(
 < Main />,
  document.getElementById('react')
 );
};
$(documentReady);
```

现在我们定义了 React 响应的基本结构。由于我们要构建包含多个视图的应用程序，因此需要一个工具，把所有的静态文件（包括 JavaScript、图片、字体和 CSS）打包起来，形成一个独立的文件。

Webpack 就是一个干这件事的工具。

我们在前面安装依赖的时候就在 `package.json` 中定义了 Webpack 包，所以现在该工具应该是可用的。

Webpack 需要一个入口文件，可以是一个`.js` 文件，用来读取子组件并将它们转换成一个独立的`.js` 文件。

我们需要为 Webpack 定义一个配置，帮助识别用于生成单个`.js` 文件的入口文件和加载器。此外，需要为生成的代码定义文件名。

我们的 Webpack 配置如下：

```
module.exports = {
  entry: "./static/main.js",
  output: {
    path: __dirname + "/static/build/",
    filename: "bundle.js"
 },
resolve: {
    extensions: ['', '.js', '.jsx']
 },
```

```
    module: {
        loaders: [
            { test: /\.js$/,exclude: /node_modules/, loader: "babel-
            loader ", query:{presets:['react','es2015']}
            }]
        }
};
```

你可以基于自己的需求扩展该配置。有的开发者可能会使用`*.html`文件作为入口，那样的话需要对该配置做一些更改。

使用下面的命令构建第一个 Web 视图：

```
$ webpack -d
```

使用`-d`参数开启调试模式，这将生成一个名为`bundle.js.map`的文件，其中包含了 Webpack 的所有活动信息。

因为我们将会多次重复构建应用，所以使用`--watch`或`-w`标志，以持续监听`main.js`文件的变更。

至此我们的 Webpack 命令如下所示：

```
$ webpack -d -w
```

应用程序构建后记得更改`app.py`中的路由信息：

```
@app.route('/index')
def index():
  return render_template('index.html')
```

现在主页应该如下图所示。



你可以使用检查（Inspect）模式来检查 React 和 react-dom 是否在后台运行。

这是了解 React 工作原理的一个非常基本的结构。再来看下我们的例子，我们构建了推文 Web 视图，用户也可以查看旧的推文。

创建 `Tweet.js` 文件，其中应该包括推文的基本结构，如用户输入内容的文本框以及发送按钮。在 `Tweet.js` 文件中输入如下代码：

```
export default class Tweet extends React.Component {

render(){
 return(
    <div className="row">
            </nav>
    <form >
      <div >
        <textarea ref="tweetTextArea" />
        <label>How you doing?</label>
          <button >Tweet now</button>
      </div>
     </form>
    </div>
   );
  }
}
```

如下更改 `render` 函数，从 `main.js` 调用 `render` 函数以将其加载到主页上：

```
import Tweet from "./components/Tweet";
render(){
  return (
  <div>
    <Tweet />
  </div>
 );
}
```

如果你加载该页面，将会看到十分简陋的页面。而我们想要创建一个引入注目的 Web 应用程序，所以应该在这里引入几个 CSS。我们使用 Materialize CSS（`http://materializecss.com/getting-started.html`）。

在 `index.html` 中增加如下代码块：

```
<link rel="stylesheet"
  href="https://cdnjs.cloudflare.com/ajax/libs/
  materialize/0.98.1/css/materialize.min.css">
<script src="https://cdnjs.cloudflare.com/ajax/libs/
  materialize/0.98.1/js/materialize.min.js"></script>
```

同时还需要更新 `Tweet.js` 文件为如下所示：

```
render(){
 return(
     <div className="row">
      <form >
       <div className="input-field">
         <textarea ref="tweetTextArea" className="materialize-textarea" />
         <label>How you doing?</label>
           <button className="btn waves-effect waves-light
             right">Tweet now <i className="material-icons
             right">send</i></button>
       </div>
      </form>
     </div> );
}
```

尝试发送几条推文，随后推文就会显示在页面上。

在 main.js 的主类中，添加如下构建函数来初始化状态：

```
constructor(props){
 super(props);
 this.state =  { userId: cookie.load('session') };
 this.state={tweets:[{'id': 1, 'name': 'guest', 'body': '"Listen to
 your heart. It knows all things." - Paulo Coelho #Motivation' }]}
}
```

更新 render 函数：

```
render(){
  return (
  <div>
     <TweetList tweets={this.state.tweets}/>
  </div>
  );
 }
}
```

创建 TweetList.js 文件来显示推文，该文件中的代码如下所示：

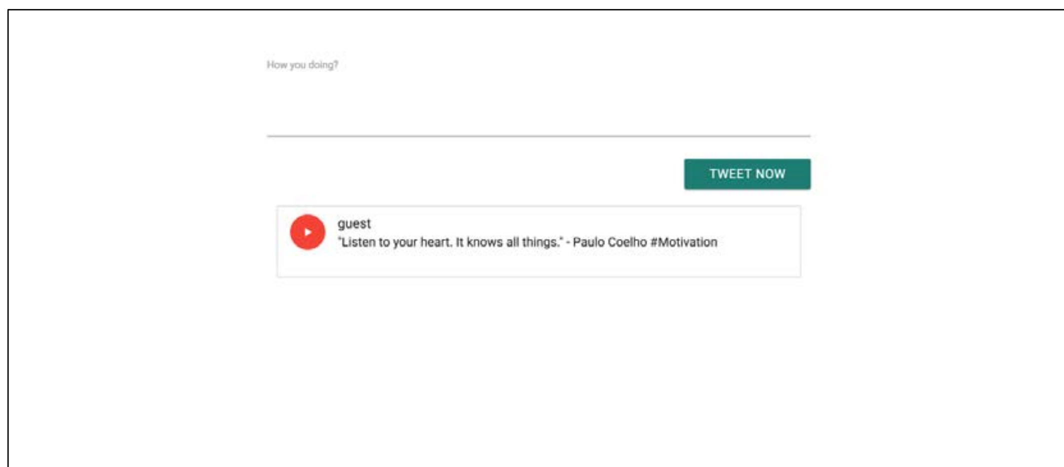
```
export default class TweetList extends React.Component {
render(){
   return(
   <div>
     <ul className="collection">
      <li className="collection-item avatar">
      <i className="material-icons circle red">play_arrow</i>
      <span className="title">{this.props.tweetedby}</span>
     <p>{this.props.body}</p>
     <p>{this.props.timestamp}</p>
      </li>
     </ul>
    </div>
   );
```

```
    }
  }
```

添加了该模板后再检查页面看看 CSS 是否生效。但是在此之前我们使用 Webpack 来构建，请确保添加了下面这行，且在每次构建的时候都要加载 bundle.js——用于在 index.html 文件中运行 Web 视图。

```
<script type="text/javascript" src="./static/build/bundle.js"></script>
```

现在主页应该看起来如下图所示。



现在开始发送推文——发送更新的推文后，这些推文应该同时被更新到 TweetList.js 中。

更新 Tweet.js 的代码，以便将推文发送到 main.js 中来处理。现在我们需要将推文发送到 main.js，那么需要修改 Tweet.js 中的以下代码：

```
sendTweet(event){
  event.preventDefault();
  this.props.sendTweet(this.refs.tweetTextArea.value);
  this.refs.tweetTextArea.value = '';
}
```

另外，务必使用 `form onSubmit` 属性更新 `render` 函数，如下所示：

```
<form onSubmit={this.sendTweet002Ebind(this)}>
```

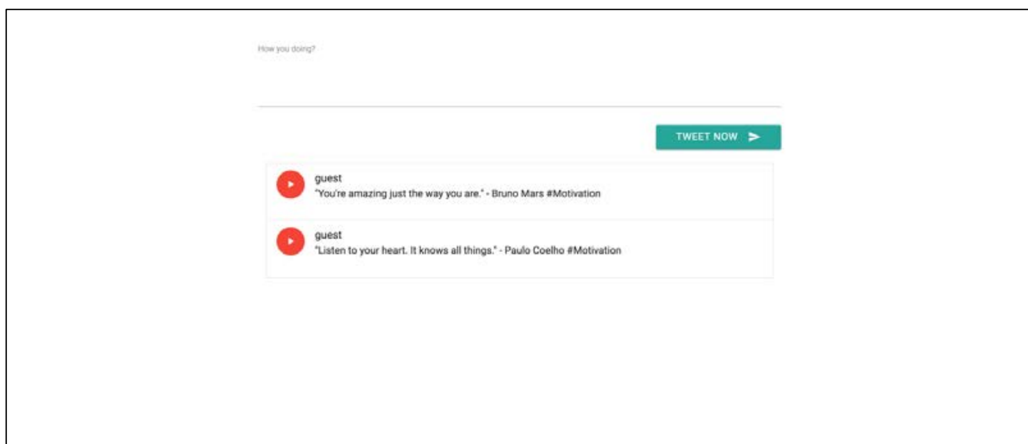
然后，在文本区域内添加内容并提交推文。现在我们来更新 `main.js` 的 `render` 函数以发送新的推文，如下所示：

```
<Tweet sendTweet={this.addTweet.bind(this)}/>
```

另外还需要在 `Main` 类中添加 `addTweet` 函数，该函数定义如下：

```
addTweet(tweet):
 let newTweet = this.state.tweets;
 newTweet.unshift({{'id': Date.now(), 'name': 'guest','body': tweet})
this.setState({tweets: newTweet})
```

发送了新的推文后，页面会如下图所示。



到目前为止，我们使用 React 将数据保存到数组中。由于我们已经建立了用于保存数据的微服务，所以我们应该将 Web 视图与后台服务整合起来。

在微服务中集成 Web 视图

为了将我们的微服务与 Web 视图集成，我们使用 AJAX 进行 API 调用。需要在 `main.js` 中添加以下代码片段来拉取整个推文列表：

```
componentDidMount() {
  var self=this;
  $.ajax({url: `/api/v2/tweets/`,
  success: function(data) {
    self.setState({tweets: data['tweets_list']});
    alert(self.state.tweets);
    return console.log("success");
   },
 error: function() {
  return console.log("Failed");
  }
});
```

类似地，修改 `main.js` 中的 `addTweet` 方法，如下所示：

```
addTweet(tweet){
  var self = this;
  $.ajax({
    url: '/api/v2/tweets/',
    contentType: 'application/json',
    type: 'POST',
    data: JSON.stringify({
      'username': "Agnsur",
    'body': tweet,
     }),
     success: function(data) {
          return console.log("success");
     },
     error: function() {
       return console.log("Failed");
     }
   });
  }
```

由于多条推文将会重复使用类似的推文模板，我们来创建另一个名为 `templatetweet.js` 的组件，其中包含以下代码：

```
export default class Tweettemplate extends React.Component {
 render(props){
  return(
  <li className="collection-item avatar">
    <i className="material-icons circle red">play_arrow</i>
    <span className="title">{this.props.tweetedby}</span>
    <p>{this.props.body}</p>
    <p>{this.props.timestamp}</p>
  </li>

  );
 }
}
```

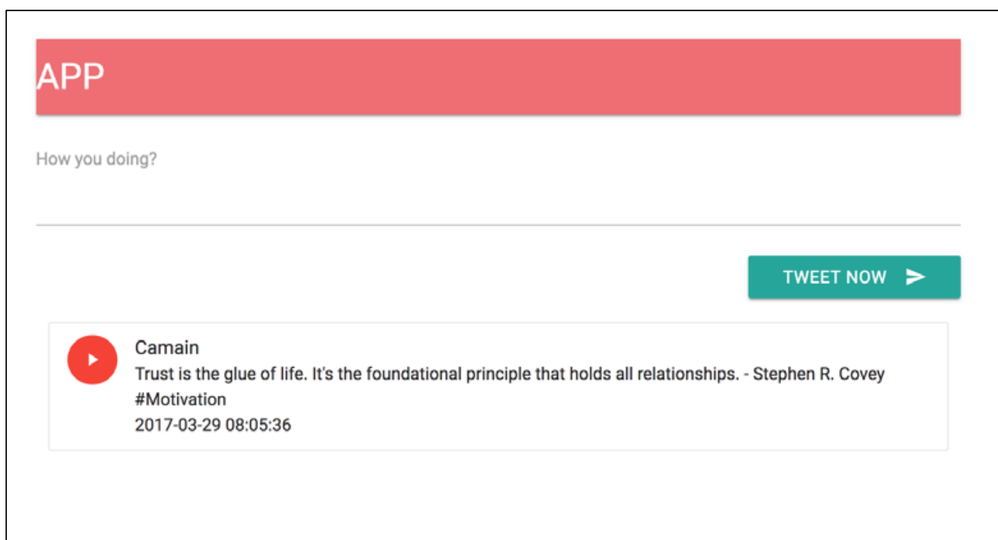
记住，我们已经根据数据库的集合键值更改了 props 域。

另外，更新 `TweetList.js` 文件以使用上述模板：

```
import Tweettemplate from './templatetweet'

export default class TweetList extends React.Component {
render(){
 let tweetlist = this.props.tweets.map(tweet => <Tweettemplate key=
 {tweet.id} {...tweet} />);
return(
    <div>
      <ul className="collection">
        {tweetlist}
      </ul>
    </div>
  );
 }
}
```

现在的主页应该如下图所示。



用户验证

所有推文都应受到保护，并且只展示给我们想要向其展示的受众。此外，不允许匿名用户发送推文。为此，需要创建一个数据库和网页，以便新用户登录到推文 Web 视图。请记住，我们将使用 Flask 来验证用户，并将数据发布给后端用户。

用户登录

创建用户登录页面模板，已存在的用户可以通过输入用户名和密码来登录，如下面代码片段所示：

```
<form action="/login" method="POST">
 <div class="login">
 <div class="login-screen">
 <div class="app-title">
  <h1>Login</h1>
 </div>
 <div class="login-form">
 <div class="control-group">
```

```
      <input type="text" class="login-field" value=""
       placeholder="username" name="username">
      <label class="login-field-icon fui-user" for="login-name"></label>
    </div>

  <div class="control-group">
      <input type="password" class="login-field" value=""
       placeholder="password" name="password">
      <label class="login-field-icon fui-lock" for="login-pass"></label>
  </div>
    <input type="submit" value="Log in" class="btn btn-primary btn-
    large btn-block"><br>
    Don't have an account? <a href="{{ url_for('signup') }}">Sign up
    here</a>.
    </div>
```

将数据发送到在 app.py 文件中定义的登录页面。

但首先，检查 session 是否存在；如果不存在，那么用户将被重定向到登录页面。在 app.py 中添加以下代码，验证用户 session 的详细信息：

```
@app.route('/')
def home():
  if not session.get('logged_in'):
     return render_template('login.html')
  else:
     return render_template('index.html', session =
  session['username'])
```

创建登录路由，并验证用户的登录凭证来授权用户发送推文。

下面是代码片段：

```
@app.route('/login', methods=['POST'])
def do_admin_login():
  users = mongo.db.users
  api_list=[]
  login_user = users.find({'username': request.form['username']})
  for i in login_user:
```

```
    api_list.append(i)
  print (api_list)
  if api_list != []:
     if api_list[0]['password'].decode('utf-8') ==
     bcrypt.hashpw(request.form['password'].encode('utf-8'),
     api_list[0]['password']).decode('utf-8'):
         session['logged_in'] = api_list[0]['username']
         return redirect(url_for('index'))
     return 'Invalid username/password!'
   else:
     flash("Invalid Authentication")

return 'Invalid User!'
```

完成后，访问 URL 根路径，将出现登录界面，如下图所示。

如图所示，我们设置了一个 **Sign up now** 注册链接，用来创建一个新用户。

请记住，我们正在使用 API 为数据库中的用户集合中的用户授权。

用户注册

创建一个注册页面来让新用户注册，然后发送推文。

创建 `signup.html` 以请求用户信息，如以下代码片段所示：

```
<div class="container">
 <div class="row">
   <center><h2>Sign up</h2></center>
     <div class="col-md-4 col-md-offset-4">
        <form method=POST action="{{ url_for('signup') }}">
            <div class="form-group">
                <label>Username</label>
                <input type="text" class="form-control"
                  name="username" placeholder="Username">
            </div>
            <div class="form-group">
                <label>Password</label>
                <input type="password" class="form-control"
                name="pass" placeholder="Password">
            </div>
            <div class="form-group">
                <label>Email</label>
                <input type="email" class="form-control"
                name="email" placeholder="email">
            </div>
            <div class="form-group">
                <label>Full Name</label>
                <input type="text" class="form-control"
                name="name" placeholder="name">
            </div>
            <button type="submit" class="btn btn-primary btn-
               block">Signup</button>
         </form>
         <br>
      </div>
    </div>
</div>
```

该代码是后台 API 向用户提交数据的模板。

我们创建一个注册路由，使用 `GET` 和 `POST` 方法来读取页面，并将数据提交到后端数据库。以下是需要添加到 `app.py` 中的代码片段：

```
@app.route('/signup', methods=['GET', 'POST'])
def signup():
  if request.method=='POST':
    users = mongo.db.users
    api_list=[]
    existing_user = users.find({'$or':
    [{"username":request.form['username']} ,
     {"email":request.form['email']}]})
        for i in existing_user:
          api_list.append(str(i))
        if api_list == []:
          users.insert({
          "email": request.form['email'],
          "id": random.randint(1,1000),
          "name": request.form['name'],
          "password": bcrypt.hashpw(request.form['pass'].
            encode('utf-8'), bcrypt.gensalt()),
          "username": request.form['username']
        })
        session['username'] = request.form['username']
        return redirect(url_for('home'))

     return 'That user already exists'
   else :
     return render_template('signup.html')
```

用户注册后，将会设置 session 并重定向到主页。

Sign up 页面应如下图所示。

Sign up

Username

Username

Password

Password

Email

email

Full Name

name

Signup

我们已经验证了用户，但如果用户想更新自己的个人信息该怎么办？为此，需要创建个人资料页面。

用户资料

创建个人资料页面（profile.html），已登录的用户可以通过导航面板访问该页面。

添加如下代码到 profile.html 中：

```
<div class="container">
 <div class="row">
   <center><h2>Profile</h2></center>
     <div class="col-md-4 col-md-offset-4">
        <form method=POST action="{{ url_for('profile') }}">
            <div class="form-group">
                <label>Username</label>
                <input type="text" class="form-control"
                 name="username" value='{{username}}'>
            </div>
            <div class="form-group">
                <label>Password</label>
                <input type="password" class="form-control"
                 name="pass" value='{{password}}'>
            </div>
```

```
                <div class="form-group">
                    <label>Email</label>
                    <input type="email" class="form-control"
                     name="email" value={{email}}>
                </div>
                <div class="form-group">
                    <label>Full Name</label>
                    <input type="text" class="form-control"
                     name="name" value={{name}}>
                </div>
                <button type="submit" class="btn btn-primary btn-
                 block">Update</button>
            </form>
          <br>
       </div>
   </div>
</div>
```

由于我们创建了个人资料页面，因此需要为其创建一条路由，该路由将读取数据库以获取用户详细信息，并将其 POST 回数据库。

app.py 中的代码片段如下所示：

```
def profile():
   if request.method=='POST':
     users = mongo.db.users
     api_list=[]
     existing_users = users.find({"username":session['username']})
     for i in existing_users:
       api_list.append(str(i))
     user = {}
     print (api_list)
     if api_list != []:
       print (request.form['email'])
       user['email']=request.form['email']
       user['name']= request.form['name']
       user['password']=request.form['pass']
       users.update({'username':session['username']},{'$set':
```

```
          user} )
        else:
            return 'User not found!'
        return redirect(url_for('index'))
    if request.method=='GET':
      users = mongo.db.users
      user=[]
      print (session['username'])
      existing_user = users.find({"username":session['username']})
      for i in existing_user:
          user.append(i)
      return render_template('profile.html', name=user[0]['name'],
      username=user[0]['username'], password=user[0]['password'],
      email=user[0]['email'])
```

添加了最后一段代码后，你的个人资料页面应该如下图所示。

Profile

Username

Upostaing

Password

Email

ColleenSLam@dayrep.com

Full Name

Colleen

Update

此外，我们应该在导航模板中的 `Tweet.js` 文件中添加个人资料链接，方法是在其中添加以下两行：

```
<li><a href="/profile">Profile</a></li>
<li><a href="/logout">Logout</a></li>
```

现在你的主页看起来应如下图所示。



用户注销

可以看出，在上一节中，我们提供了注销的路由，来删除用户 session，并将用户重定向到登录页面。以下是来自 app.py 的代码片段：

```
@app.route("/logout")
def logout():
  session['logged_in'] = False
  return redirect(url_for('home'))
```

现在我们的应用程序已经完全建立起来了，从用户登录开始，到发送推文再到最后注销。

测试 React webViews

构建了 Web 视图后，需要对其进行测试。此外，测试也可以帮助你构建更好的代码。

有一些基于图形界面的测试框架可以帮助你测试 Web 应用程序。下面我们讨论其中的两个。

Jest

Jest 是一款由 Facebook 开发的用来测试 JavaScript 的单元测试框架，其用于测试各个组件。它使用起来非常简单，是一个标准化的可独立运行的框架。

它基于模拟的 DOM 实现测试组件，并运行不同的测试来检查功能。它自动解析依赖关系。此外，你还可以并行运行所有测试。

参考以下链接的介绍了解如何为 React 应用程序编写测试用例：

`https://facebook.github.io/jest/docs/tutorial-react.html`

Selenium

Selenium 是一款用于测试 Web 应用程序的开源和可移植的自动化软件测试工具。它提供端到端测试，也就是说该测试针对真实浏览器执行测试，以便测试多层应用程序的整个堆栈。

它具有以下组件。

- **IDE**：描述测试工作流。
- **Selenium WebDriver**：自动执行浏览器测试。将命令直接发送到浏览器并接收结果。
- **Selenium RC**：远程控制用户创建测试用例。
- **Grid**：在不同浏览器中并行运行测试用例。

这是测试 Web 应用程序的最佳工具之一，强烈推荐。

关于 Selenium 的更多信息请访问 `http://www.seleniumhq.org/docs/`。

本章小结

本章的重点是创建前端 Web 视图以及改进它们来吸引消费者。在本章中我们还了解了如何使用 React 构建 Web 视图并实现与后端服务的交互。下一章的内容将会更有趣，我们将阐述如何使用 Flux 来扩展应用程序，从而处理来自互联网的大量请求。

6

使用 Flux 来构建 UI 以应对大规模流量

在上一章中，我们为应用程序创建了 Web 视图，并将应用程序的前后端集成起来，这是非常重要的。

在本章中，我们将着重于构建前端部分。在理想情况下，每个模块应该只负责一件事情。在所有主要组件中，我们在单个模块上的操作过多。除了呈现不同的视图之外，还要编码向端点发出 API 请求，接收、处理和格式化响应。

本章包括以下主要内容：

- 了解 Flux
- 在 React 中实现 Flux

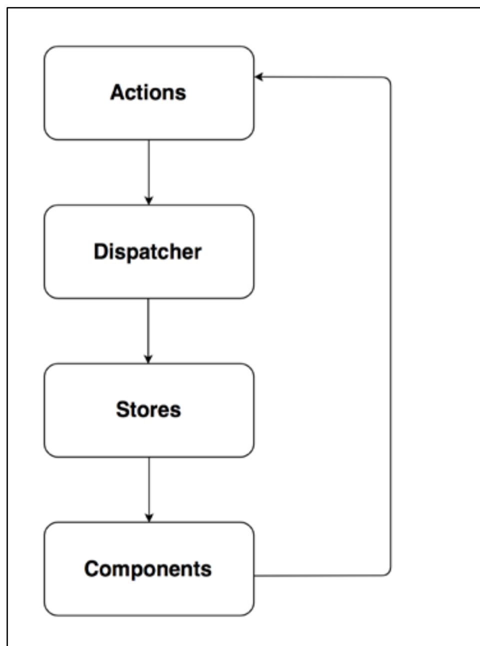
Flux 介绍

Flux 是由 Facebook 创建的一种模式，用来构建一致、稳定的 Web 应用程序。React 不能管理数据；相反，只是通过 props 和组件来接收数据，然后由组件来处理。

React 库没有告诉你该如何获取组件，或者在哪里存储数据，这是它被称为视图层的原因。在 React 中，没有像 Angular 或 Backbone 这样的框架。这就是 Flux 存在的原因。Flux 并不是一个框架，而是一种建立视图的模式。

什么是 Flux 模式？我们的各种 React 组件，例如 Tweet 组件，这些组件在 Flux 模式中做两件事情——它们执行操作或者监听数据存储。在我们的例子中，如果用户想要发布推文，则组件需要执行动作，这些动作然后与存储交互，将模式更新到 API，并对组件做出

响应。下图是 Flux 模式的视图。



Flux 概念

在继续阅读后面的内容之前，需要先了解下面的这些 Flux 概念。

- **动作（Action）**：组件通过动作与 API 进行交互并更新。在我们的例子中使用它发送推文。动作被推送到分派器后可能产生多个动作。
- **分派器（Dispatcher）**：将流入的所有事件分别派发给每个订阅者，通常是数据存储。
- **数据存储（Store）**：这是 Flux 的重要组成部分。组件时刻监听数据存储的变化。假如你发送了一条新的推文，那这就是一个动作，只要数据存储中有推文更新，就会触发一个事件，此时组件意识到必须用最新的数据更新。在 AngularJS 圈子中数据存储是一种服务，如果你使用的是 Backbone.js，那么数据存储只不过是一个集合。
- **组件（Component）**：用于保存动作的名称。



我们将使用 JSX 扩展名而不再使用 JS 扩展名，其实这没有太大区别——JS 是标准的 JavaScript 格式，而 JSX 类似 HTML 语法。你可以使用 React 轻松创建 React 组件。

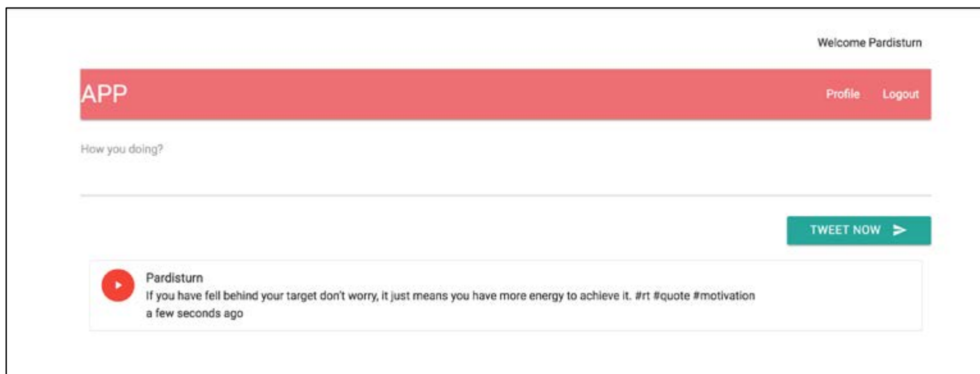
在 UI 中添加日期

在深入了解 Flux 之前，我们先在视图上添加日期功能。在此之前，你看到的推文的日期是以 TZ 格式存储在数据库中的；然而，在理想情况下，应该参考和比较当前时间。

为此，需要修改 main.jsx 文件，以便格式化推文。将以下代码添加到 main.jsx 中：

```
updatetweets(tweets){
    let updatelist = tweets.map(tweet => {
     tweet.updatedate = moment(tweet.timestamp).fromNow();
    return tweet;
  });
}
```

完成了！现在的推文看起来如下图所示。



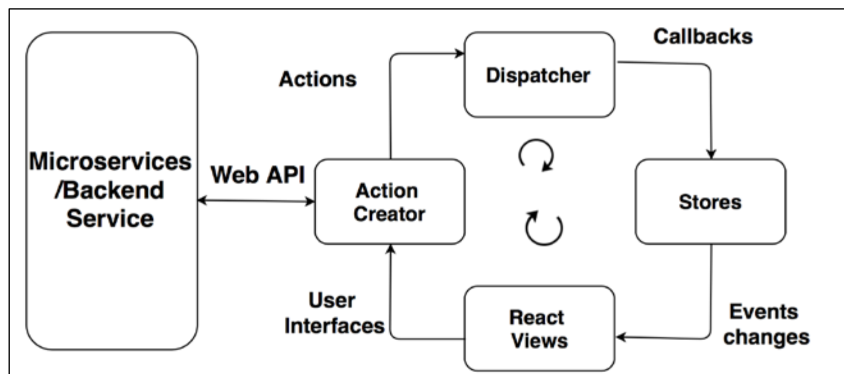
使用 Flux 创建 UI

在 Flux 中，需要给每个模块定义职责，而且必须是单一职责。React 的作用是在数据发生变化时重新渲染视图，这对我们有好处。我们要做的就是监听 Flux 的数据事件来管理数据。

Flux 不仅可以将不同的模块的职责分离，还可以在应用程序中实现单向数据流，因此 Flux 十分受欢迎。

在 Flux 循环中，对于每个模块，总是沿着一个方向运行。这样的流程设计是为了使 Flux 应用程序易于设计、扩展、管理和维护。

下面是 Flux 的架构图。



该图参考自 Flux 代码仓库（`https://github.com/facebook/flux`）。

动作和分派器

开始使用 Flux 都要选择一个起点。笔者觉得从动作（action）开始比较好。你必须选择一个流向，选择顺时针或逆时针都可以。顺时针可能是一个比较好的起点，所以我们选择该流向。

不要忘了安装 Flux 库，使用下面的命令：

```
$ npm install flux --s0061ve
```

请注意，该命令应该在应用程序目录中执行，也可以在 `package.json` 中添加该命令，执行 `npm install` 来安装软件包。

现在，我们以动作为出发点并遵循单一职责原则，创建一个动作库与 API 通信，另一个动作与分派器进行通信。

我们先在 static 目录中创建 `action` 文件夹。

我们需要执行两个操作：列出推文和添加新的推文。我们首先列出推文。使用 getAllTweets 函数创建 Tactions 文件，该函数调用 REST API 来获取所有推文，如下所示：

```
export default{
 getAllTweets(){
 //API calls to get tweets.
 }
}
```

基于 Flux 的应用程序设计很容易，对吧？因为我们知道该动作模块只有单一职责的单向数据流，或者提供 API 调用，或者调用处理该应用程序 API 请求的模块。

修改 Tactions.jsx 文件为如下所示：

```
import API from "../API"
 export default{
  getAllTweets(){
   console.log(1, "Tactions for tweets");
    API.getAllTweets();
  },
}
```

可以看到，这里引入了 API 模块，用来调用 API 获取推文。

在 static 目录下创建 API.jsx，使用以下代码片段从后端服务获取推文：

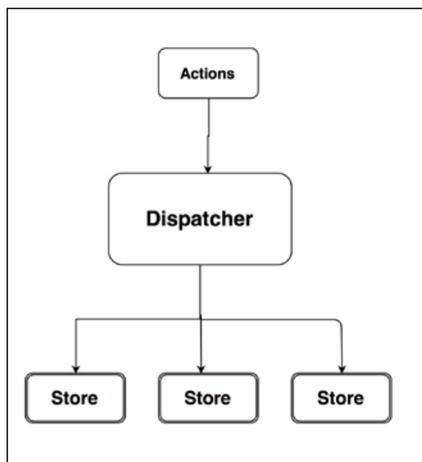
```
export default{
  getAllTweets(){
   console.log(2, "API get tweets");
   $.getJSON('/api/v2/tweets', function(tweetModels) {
      var t = tweetModels
    // We need to push the tweets to Server actions to dispatch
    further to stores.
   });
}
```

在 actions 目录下创建 Sactions 文件以请求分派器并定义 actionType：

```
    export default{
      receivedTweets(rawTweets){
       console.log(3, "received tweets");
      //define dispatcher.
     }
}
```

可以看到，我们仍然需要定义分派器。幸运的是，Facebook 创建的 Flux 中已经包含了分派器。

如前所述，Dispatcher 是应用程序的中心枢纽，它分派已注册回调的动作和数据。可以参考下图来了解数据流。



创建一个名为 `dispatcher.jsx` 的文件，使用下列代码创建一个分派器实例：

```
import Flux from 'flux';

export default new Flux.Dispatcher();
```

现在可以在应用程序中的任何地方引入分派器。

修改 `Sactions.jsx` 文件，其中 `receiveTweets` 函数的代码如下所示：

```
import AppDispatcher from '../dispatcher';
receivedTweets(rawTweets){
  console.log(3, "received tweets");
```

```
    AppDispatcher.dispatch({
      actionType: "RECEIVED_TWEETS",
       rawTweets
    })
  }
```

在 receivedTweets 函数中需要描述三件事。首先，将从 API.jsx 中的 getAllTweets 函数接收 rawTweets，我们需要做如下更新：

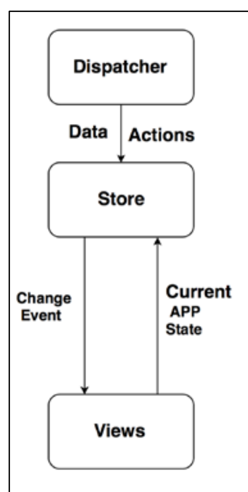
```
import SActions from './actions/SActions';

getAllTweets(){
  console.log(2, "API get tweets");
  $.getJSON('/api/v2/tweets', function(tweetModels) {
     var t = tweetModels
     SActions.receivedTweets(t)
  });
```

数据源

数据源（Store）通过控制应用程序中数据的状态来管理应用程序，也就是说数据源管理数据，通过方法检索数据，分派器回调数据。

可参考下图来理解。



现在我们已经定义了分派器，接下来需要识别分派器提供的用来接收变更的订阅者。

在 static 目录下再创建一个 stores 目录，保存所有的数据源定义。

创建一个 TStore 文件用来订阅分派器发出的所有更改。在 TStore 文件中增加如下代码：

```
import AppDispatcher from "../dispatcher";

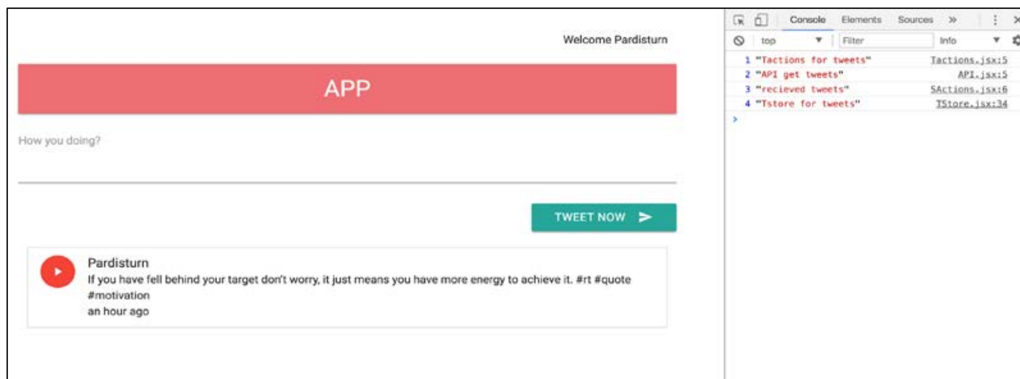
AppDispatcher.register(action =>{
 switch (action.actionType) {
 Case "RECEIVED_TWEETS" :
console.log(4, "Tstore for tweets");
 break;
  default:
 }
});
```

至此我们已经启动了推文动作，向 API 模块发送一个消息以获取所有的推文。API 执行该操作，然后调用服务器操作将数据传递给分派器。分派器标记数据并发送。我们还创建了从分派器请求数据并管理数据的数据源。

现在数据源还没有关联到 app。数据源应该在事件发生的时候产生变化，并基于此来改变视图。

因此，我们的主要组件只关心数据源发出的变化事件。现在，我们导入数据源。

在这之前我们先看看当前的应用程序是否可以正常运行，应该如下图所示。





每当应用程序达到某个稳定状态后检查用户界面的状态，这是一个好习惯。

我们继续吧。目前我们只是发布了推文，接下来我们需要确定怎样处理这些推文。所以，我们首先接收推文，然后对相应的视图发出更改事件。我们之前是使用发射器（Emitter）来做的。

发射器是我们之前使用 npm 安装的事件库的一部分。所以我们直接导入。注意，它并不是默认导出的，其上有一个 destructed 属性。然后数据源就会成为推文 EventEmitter 类的一个实例。

修改 TStore.jsx 文件，如下所示：

```
import { EventEmitter } from "events";

let _tweets = []
 const CHANGE_EVENT = "CHANGE";

 class TweetEventEmitter extends EventEmitter{
 getAll(){
   let updatelist = _tweets.map(tweet => {
     tweet.updatedate = moment(tweet.timestamp).fromNow();
    return tweet;
   });
 return _tweets;
 }
 emitChange(){
   this.emit(CHANGE_EVENT);
 }

 addChangeListener(callback){
  this.on(CHANGE_EVENT, callback);
 }
 removeChangeListener(callback){
   this.removeListener(CHANGE_EVENT, callback);
```

```
     }
    }
    let TStore = new TweetEventEmitter();

    AppDispatcher.register(action =>{
     switch (action.actionType) {
       case ActionTypes.RECEIVED_TWEETS:
         console.log(4, "Tstore for tweets");
         _tweets = action.rawTweets;
         TStore.emitChange();
       break;
      }
     });
    export default TStore;
```

哇，代码太多了，太难理解了！下面我们一个部分一个部分来讲解。

首先，我们将使用下面的导入工具从事件包中导入 EventEmitter 库：

```
import { EventEmitter } from "events";
```

接下来，把接收到的推文保存在_tweets 中并在 getAll()函数中更新推文，以便在视图中显示基于当前时间的推文时间线。

```
getAll(){
  let updatelist = _tweets.map(tweet => {
      tweet.updatedate = moment(tweet.timestamp).fromNow();
      return tweet;
    });
  return _tweets;
}
```

我们还创建了用于添加和删除更改事件侦听器视图的函数。这两个函数也将只是对 EventEmitter 语法的封装。

这些函数获得由视图发送的 callback 参数。它们主要用于添加或删除监听器，以便视图开始或停止监听数据源中的更改。将以下代码添加到 TStore.jsx 中：

```
addChangeListener(callback){
```

```
    this.on(CHANGE_EVENT, callback);
  }
  removeChangeListener(callback){
   this.removeListener(CHANGE_EVENT, callback);
  }
```

确保所有更新代码的控制台中没有错误。

接下来是视图，我们创建一个从存储中提取数据的函数，并为组件的状态创建一个对象。

在 main.jsx 中编写 getAppState()函数以保存应用程序的状态，代码如下所示：

```
let getAppState = () =>{
  return { tweetslist: TStore.getAll()};
}
```

如前所述，文件的扩展名是.js 还是.jsx 没有关系。

现在我们从 Main 类中调用此函数，并调用 main.jsx 中创建的添加和删除监听器的函数。使用以下代码：

```
import TStore from "./stores/TStore";

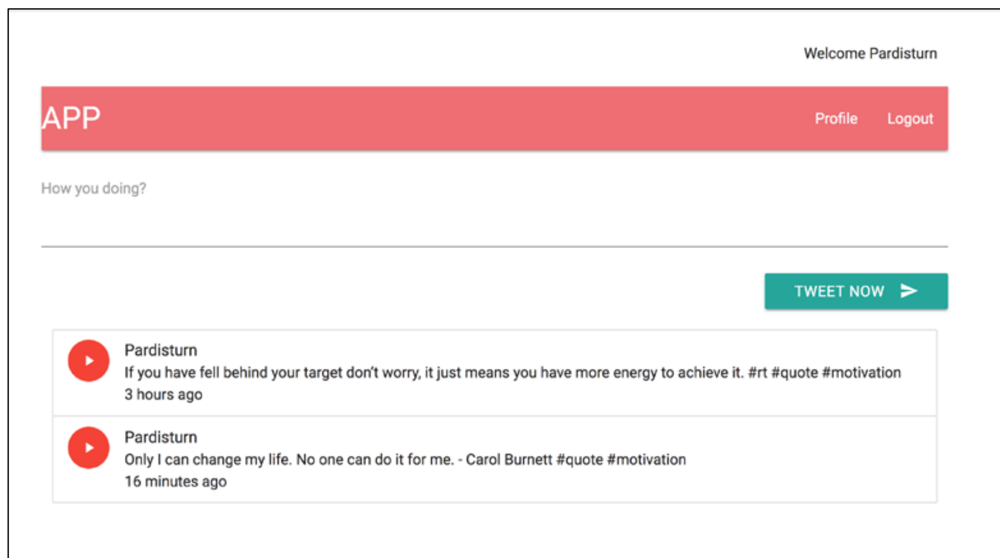
class Main extends React.Component{
  constructor(props){
   super(props);
   this.state= getAppState();
   this._onChange = this._onChange.bind(this);
   //defining the state of component.
  }
// function to pull tweets
  componentDidMount() {
  TStore.addChangeListener(this._onChange);
 }
componentWillUnMount() {
  TStore.removeChangeListener(this._onChange);
}
```

```
_onChange(){
 this.setState(getAppState());
 }
```

此外，我们必须更新 `render` 函数以获取视图中的 `Tweetslist` 状态。使用以下代码片段完成：

```
render(){
  return (
   <div>
   <Tweet sendTweet={this.addTweet.bind(this)}/>
      <TweetList tweet={this.state.tweetslist}/>
   </div>
   );
}
```

很好，做了这么多工作后，页面上显示的推文应该没有任何问题了，如下图所示。



我们的应用程序运行得很好。

如果你还记得 Flux 的架构图，应该知道我们已经完成了 Flux 的整个流程，但是还需要创建用户添加推文的 API 的流程。

使用 Flux 来实现发送推文的功能，在 `main.jsx` 中做一些改动。在 `render` 函数中，Tweet 调用下面这行中的函数：

```
<Tweet sendTweet={this.addTweet.bind(this)}/>
```

不带参数调用 Tweet 组件，如下所示：

```
<Tweet />
```

此外，在 Tweet 组件中，调用 `TActions` 模块来添加推文。更新 `Tweet` 组件中的代码，如下所示：

```
import TActions from "../actions/Tactions"
export default class Tweet extends React.Component {
 sendTweet(event){
  event.preventDefault();
  // this.props.sendTweet(this.refs.tweetTextArea.value);
  TActions.sendTweet(this.refs.tweetTextArea.value);
  this.refs.tweetTextArea.value = '';
 }

}
```

`Tweet` 组件中的 `Render` 函数保持不变。

新添加一个 `sendTweet` 函数来调用后端应用程序的 URL 端点，进行 API 调用，并将其添加到后端数据库。

现在的 `Taction.jsx` 文件如下所示：

```
  import API from "../API"

export default{
  getAllTweets(){
   console.log(1, "Tactions for tweets");
   API.getAllTweets();
  },
  sendTweet(body){
    API.addTweet(body);
```

```
    }
  }
```

在 `API.jsx` 中添加 `API.addTweet` 函数，用于执行 API 调用，同时更新推文列表的状态。在 `API.jsx` 文件中添加 `addTweet` 函数：

```
addTweet(body){
   $.ajax({
       url: '/api/v2/tweets',
       contentType: 'application/json',
       type: 'POST',
       data: JSON.stringify({
       'username': "Pardisturn",
       'body': body,
       }),
   success: function() {
       rawTweet => SActions.receivedTweet({ tweetedby:
       "Pardisturn",body: tweet, timestamp: Date.now})
    },
    error: function() {
           return console.log("Failed");
    }
 });
}
```

此外将新添加的推文传给服务器动作（action），以便来分派和在数据源中使用。

添加新函数 `receivedTweet`，来分派推文。使用下面的代码：

```
receivedTweet(rawTweet){
  AppDispatcher.dispatch({
    actionType: ActionTypes.RECEIVED_TWEET,
    rawTweet })
}
```

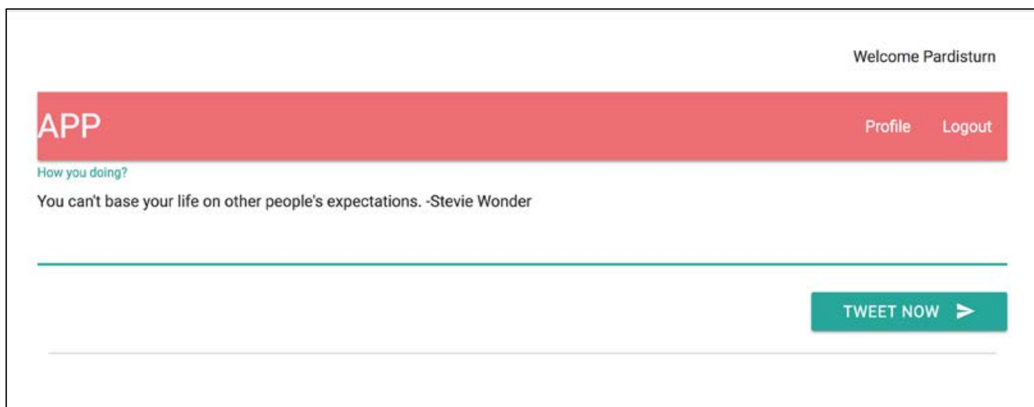
始终在 static 目录下的 `constant.jsx` 文件中定义 `ActionTypes`。

现在，我们在推文的数据源中定义 RECEIVED_TWEETActiontype，将更改发送给视图以进行进一步操作。以下是 Tstore.jsx 中修改后的 Appdispatcher.register 函数：

```
AppDispatcher.register(action =>{
 switch (action.actionType) {
      case ActionTypes.RECEIVED_TWEETS:
      console.log(4, "Tstore for tweets");
      _tweets = action.rawTweets;
      TStore.emitChange();
       break;
    case ActionTypes.RECEIVED_TWEET:
      _tweets.unshift(action.rawTweet);
      TStore.emitChange();
      break;
   default:

  }
});
```

至此，我们完成了使用 Flux 添加新推文的模块，其应该可以正常工作，如下图所示。



现在如果单击 **TWEET NOW** 按钮，应该会发出新的推文并且显示在页面上，如下图所示。



本章小结

在本章中，我们了解了如何使用 Flux 模式来构建应用程序，还了解了 Flux 的各个组件，如分派器、数据源等。Flux 提供了模块之间职责分配的良好模式。这一章你需要好好掌握，因为我们开发的是基于云平台（如 AWS、Azure 等）的应用程序，这些应用程序应该是高度响应式的。本章的主要内容是如何构建用户界面，但在下一章中，我们将了解一些重要概念，如事件溯源，以及如何通过使用不同的身份验证方式使应用程序更加安全。

7

事件溯源与 CQRS

在上一章中，我们探讨了当前业务模型的缺陷，本章我们将利用**事件溯源**（**ES**）和**CQRS**（**Command Query Responsibility Segregation，命令查询责任分离**）来克服这些障碍。

本章将讨论处理大规模可伸缩性问题的体系结构设计。我们将看到两种模式：事件溯源和 CQRS，这些模式都是用来处理巨量用户请求下的响应问题的。

许多人认为遵守十二要素法则可以让应用成为具有更高可扩展性的云原生应用程序，但是还有其他一些策略，如 ES 和 CQRS，这些策略可以使应用程序更加可靠。

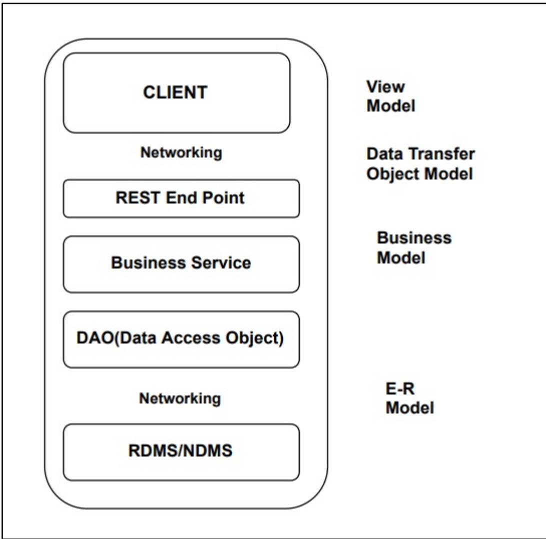
由于我们的云原生应用程序是面向互联网的，期望应对数以千计甚至数百万的请求。但仅仅实现可伸缩的基础架构，对于处理这些请求是远远不够的。应用程序需要具有极大的可扩展性。这也是这些模式出现的原因。

本章包括以下主要内容：

- 事件溯源简介
- 命令查询责任分离简介
- ES 和 CQRS 实现的示例代码
- 使用 Apache Kafka 实现事件溯源

简介

首先回顾一下分层架构，其中包括客户端、网络、业务模型、业务逻辑以及数据存储，等等。这是几乎所有的架构设计中的基本模型，如下图所示。



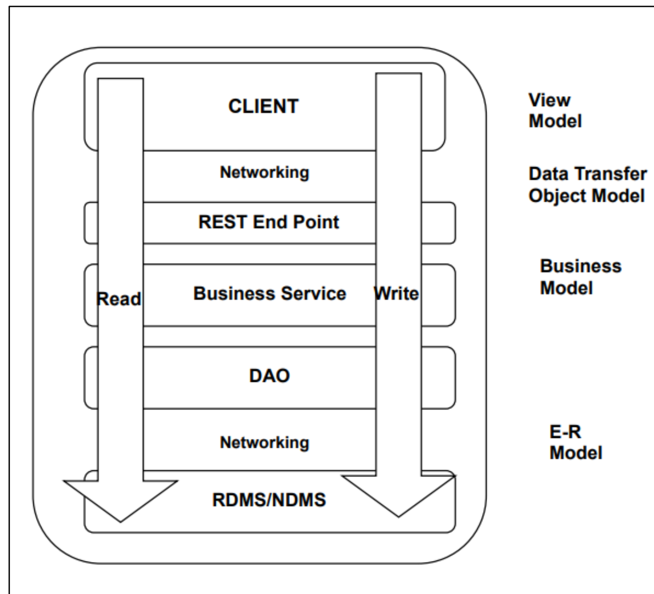
其中包括以下这些模型。

- **View Model**：用于客户端交互。
- **DTO Model**：用于客户端与 REST 端点通信。
- **Business Model**：这是 DAO（数据访问对象）和业务服务的组合，用于解释用户的请求，并与存储服务通信。
- **E-R Model**：定义了实体之间的关系（即 DTO 和 RDMS/NDMS）。

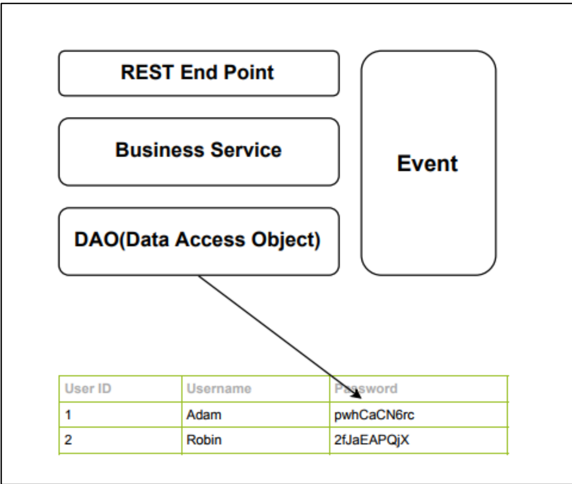
该架构的特点如下。

- **相同的应用程序堆栈**：在这个模型中，所有的读/写操作使用相同的堆栈元素，从 REST API 到业务服务，然后访问存储服务。等等，因为所有不同的组件代码都是作为一个单独的实体来部署的。

  下图显示了不同模型的**读/写**操作流程。



- **相同的数据模型**：在这种情况下，你会发现大多数时候都是使用相同或相似的数据模型进行业务逻辑处理，或读取和写入数据。
- **部署单元**：我们使用粗粒度部署单元，其中包含以下内容。
  — 构建（一个可执行的组件集合）
  — 文档（最终用户的使用帮助和发行说明）
  — 安装部件，将代码的读取和写入结合到一起
- **直接访问数据**：如果要更改数据通常是直接去做。特别是在 RDBMS 下，我们会直接更改数据，如下面这种情况——更新 **User ID** 为 1 的行数据集，我们通常直接去改。另外，一旦我们更新了该值，那么旧的值将在应用程序以及存储端失效，并且不能被检索，如下图所示。



到目前为止，我们一直在使用前面的方法，并且从用户请求的响应来看，该方法被证明是成功的。然而，还有其他的替代方法有更好的表现。

我们再来讨论上述业务架构的缺点，如下所示。

- **无法独立扩展**：由于我们的读/写操作代码驻留在相同的位置，因此无法单独扩展应用程序的读取或写入功能。假设你在特定的时间点有 90％的读取和 10％的写入，此时就无法独立扩展读取功能。为了扩展读取功能，我们需要扩展整个架构，而整个架构的其他部分又不会用到，导致资源的浪费。
- **没有历史数据**：由于在当前场景下，我们直接更新数据后，应用程序将在一段时间后开始显示最新的数据集。而且，一旦数据集被更新，就不再跟踪旧的数据值，因此旧的值将被丢弃。即使实现这样的功能也需要编写大量的代码。
- **单体方法**：这一般是单体方法，因为我们试图把它们合并在一起。而且，其中有粗粒度的部署单元，我们试图将不同组件的代码保存在一起。所以这种做法最终会导致一团糟，难以解决。

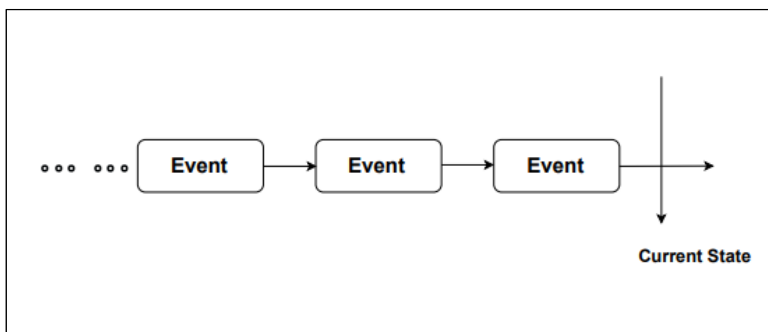
有一种方法可以解决这些问题，那就是事件溯源。

理解事件溯源

简而言之，事件溯源是一种架构模式，其通过一系列事件来确定应用程序的状态。

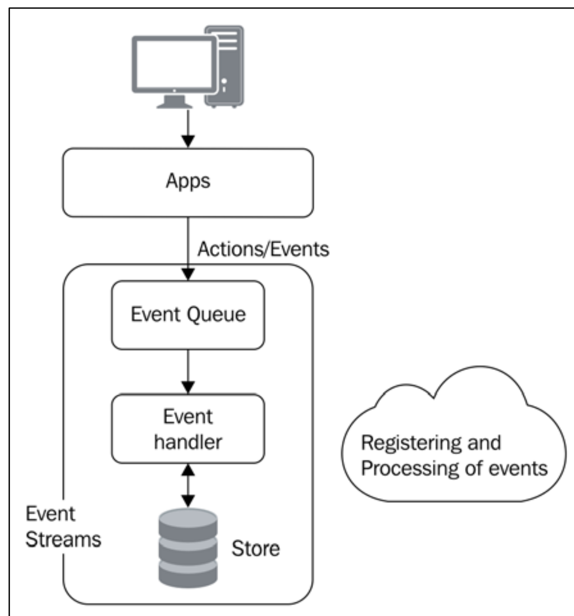
理解事件溯源的最好方法是类比。网上购物就是一个很好的例子，在这个事件处理系统中，有人下订单，在订单队列中向供应商订购系统注册。然后，在订单交货的不同阶段将此状态通知给客户。

所有这些接连发生的事件被串联成事件流，如下图所示。



所以，事件溯源会根据过去发生的事件和基于某些事务记录来进行处理。

理想的事件溯源系统基于下图所示的组件。



该图描绘了一个理想的事件处理系统，从应用程序开始到创建与某个事物相关的**事件**，然后将它们放入**事件队列**中进行进一步处理，这由**事件处理程序**来完成。基于**事件**的描述，由事件处理程序相应地处理它们，并将它们注册到**存储**中。

事件溯源遵循某些法则/原则，这使得应用程序开发过程的结构更加严谨有序。通常大多数人都会因在程序开发过程中必须遵守这些规则，而认为事件溯源复杂，因为这些原则是牢不可破的，一旦破坏这些规则就会在应用程序中造成巨大的混乱。

事件溯源定律

下面列出了在任何系统（即应用程序设计）上实施事件溯源时都需要遵循的事件溯源法则。

- **幂等**：理想的事件溯源业务逻辑必须是幂等的。这意味着，当你对输入的数据流执行业务逻辑时，应用程序的结果状态将始终保持不变。是的，这是事实，无论执行多少次业务逻辑，结果的状态都将保持不变。
- **隔离**：事件溯源不得依赖于外部事件流。这是事件溯源中最重要的原则之一。一般来说，业务逻辑很少在真空中执行。应用程序通常会与外部实体交互。而且，即使开发人员不考虑这一点，应用程序也会使用来自外部源的缓存信息。现在的问题是，如果你的业务逻辑使用外部输入来计算结果会发生什么？我们以股票交易时不断变化的股票价格为例，这意味着在多次评估的情况下，进行状态计算时的股票价格将会不同，这违反了幂等规则。



根据开发者的理解，这是一个很难满足的条件。但是，解决这个问题的方法是将通知从外部事件注入主事件流中。由于这些通知现在是主事件流的一部分，因此每次都会得到预期的结果。

- **质量保证**：开发完成的事件溯源应用程序应该经过良好测试。为事件溯源应用程序编写测试用例很容易——鉴于我们按照之前定义的原则编写测试用例，通常需要输入列表并返回一些状态。
- **可恢复**：事件溯源应用程序应支持恢复和重演。如果你的云原生应用程序符合十二要素应用的所有设计原则，那么在创建适用于云平台的应用程序时，事件溯源

将在灾难恢复中扮演着至关重要的角色。

假设事件流是持久的，事件溯源应用程序的最初优势是计算应用程序的状态。通常情况下，由于多种原因，应用程序可能会崩溃；事件溯源可以帮助我们识别应用程序的最后状态，从而使其快速恢复以减少停机时间。此外，利用事件溯源的重演功能，可查看审计的过去状态以及进行故障排除。

- **大数据**：事件溯源应用程序通常会产生大量的数据。由于事件溯源应用程序会跟踪每个事件，因此可能会生成大量的数据。这取决于事件的多少、它们发生的频率以及事件数据有效载荷的大小。
- **一致性**：事件溯源应用程序通常保持事件注册的一致性。想想银行交易——银行交易中发生的每一件事情都非常重要。应该指明，在记录时应保持一致性。

了解这些事件是在过去发生的，这非常重要，因为当我们将这些事件命名时，这些名字应该是可以理解的。有效的事件名称示例如下：

- `PackageDeliveredEvent`
- `UserVerifiedEvent`
- `PaymentVerifiedEvent`

无效的事件名称示例如下：

- `CreateUserEvent`
- `AddtoCartEvent`

以下是一些事件示例代码：

```
class ExampleApplication(ApplicationWithPersistencePolicies):
  def __init__(self, **kwargs):
    super(ExampleApplication, self).__init__(**kwargs)
  self.snapshot_strategy = None
  if self.snapshot_event_store:
      self.snapshot_strategy = EventSourcedStrategy(
          event_store=self.snapshot_event_store,
      )
  assert self.integer_sequenced_event_store is not None
  self.example_repository = ExampleRepository(
      event_store=self.integer_sequenced_event_store,
```

```
        snapshot_strategy=self.snapshot_strategy,
    )
```

你应该记住以下几点：

- 每个事件都是不可改变的，这意味着一个事件一旦被发出，就无法恢复。
- 你永远不会删除一个事件。即使删除一个事件，我们也认为删除本身也是一个事件。
- 事件流由消息代理架构驱动。消息代理包括 RabbitMQ、ActiveMQ 等。

我们来看看事件溯源的优点，如下所示：

- 事件溯源提供了快速地重演系统的能力。
- 事件溯源赋予了我们对数据的控制权，也就是说需要通过查看事件流来处理数据，例如通过审计、分析来处理。
- 通过查看事件中的数据，很容易理解在一段时间内发生了什么。
- 事件重演将有利于故障排除和错误修复。

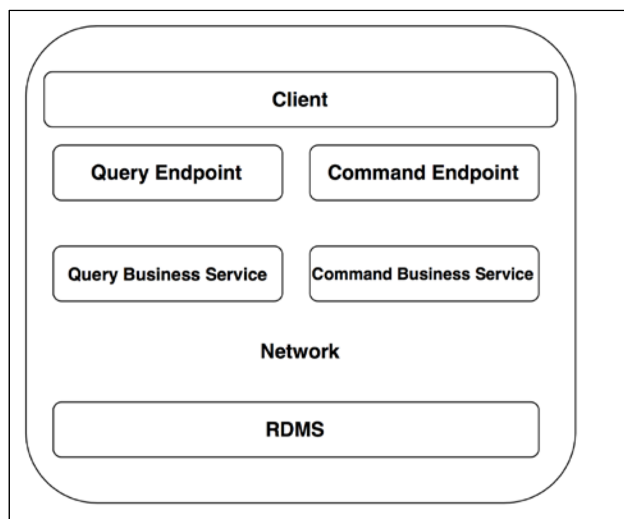
现在，问题来了，如果产生巨量的事件，是否会影响应用程序的性能呢？我会说，是的！

由于应用程序为每个需要由事件处理器处理的事务生成事件，所以应用程序的响应时间就减少了。解决这个问题的方法是使用 CQRS。

CQRS 介绍

命令查询责任隔离，这个名字听起来很新奇，它的意思是解耦系统的输入和输出。对于 CQRS 中，我们主要讨论应用程序的读/写特性；所以，在 CQRS 上下文中的命令主要用于写操作，而查询是读操作，责任意味着我们分离读和写操作。

我们看一下第一节中介绍的体系结构，然后应用 CQRS，则体系结构将被分成两半，看起来如下图所示。



下面我们来看一些代码示例。

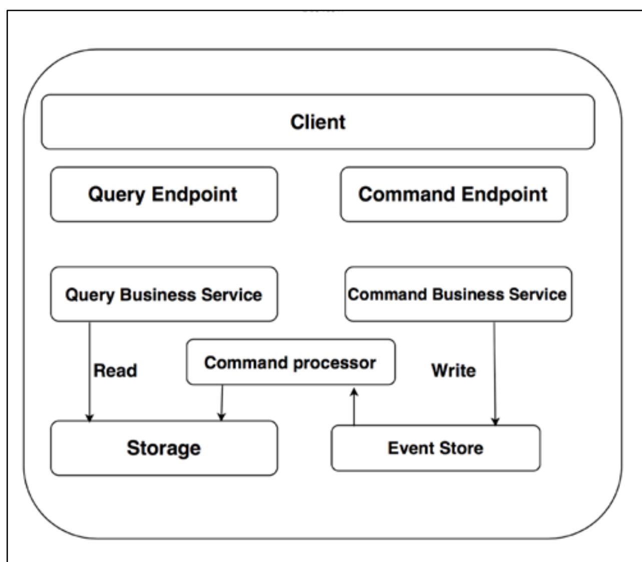
传统的接口模块看起来像这样：

```
Class managementservice(interface):
 Saveuser(userdata);
Updateuser(userid);
listuserbyusername(username);
listuserbyid(userid);
```

拆分，应用 CQRS 之后，看起来是这样：

```
Class managementcommandservice(interface):
 Saveuser(userdata);
Updateuser(userid);
Class managementqueryservice(interface):
listuserbyusername(username);
listuserbyid(userid);
```

因此，在实施了事件溯源和 CQRS 之后，总体架构将如下图所示。



在经典的单体应用程序中，有写入数据库的端点和读取数据库的端点。读取和写入操作使用相同的数据库，并且在从数据库接收到确认或提交信息之前，不会给端点回复。

在大规模的情况下，高入站事件吞吐量和复杂的事件处理对性能的要求都很高，我们不能让查询变得太慢，也不能在每次发生新的入站事件时坐下来等待它被处理完成。

读/写操作的流程如下。

- **写模式**：在这种情况下，当命令从端点被触发并被**命令业务服务**接收时，它首先向**事件存储库**发出每个事件。在**事件存储库**中，也有一个**命令处理器**，换句话说，就是**事件处理器**，这个命令处理器能够把应用程序状态导入一个单独的存储器中，该存储器可以是一个关系型存储器。
- **读模式**：在读模式下，我们只需使用查询端点来查询客户端应用程序想要读取或检索的数据。

该模式最大的优势是我们不需要经过写模式（位于上图的右侧）。在查询数据库时，这个过程使我们的查询速度更快，并缩短了响应时间，从而提高了应用程序的性能。

CQRS 架构的优点

该架构具有以下优点。

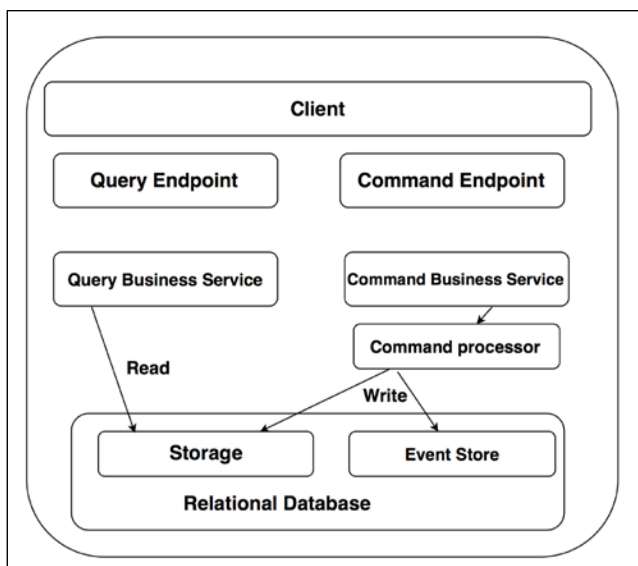
- **独立的可伸缩性和部署**：现在我们可以根据其使用情况来扩展和部署单个组件，就像微服务一样。可以为每个任务分别创建微服务，比如该架构栈中应该有一个读微服务和一个写微服务。
- **技术选择**：不同业务模式的技术选择是自由的。例如，对于命令功能，我们可以选择 Scala 或类似的（假设我们有一个复杂的业务模型，而且有很多数据要写）。对于查询，我们可以选择 ROR（Ruby on Rails）或 Python（我们已经在使用）。

这种架构最适合 **DDD（领域驱动设计）**的有界上下文，因为我们可以为微服务定义业务上下文。

事件溯源与 CQRS 面临的挑战

每一种架构设计模式实现都要面临一些挑战。事件溯源和 CQRS 要面临的挑战如下。

- **不一致**：使用事件溯源和 CQRS 开发的系统大部分是一致的。但是，当我们将**命令业务服务**发布的事件存储在**事件存储器**中时，并将应用程序的状态放在主**存储**中时，我会说这种系统是不完全一致的。如果我们真的希望使用事件溯源和 CQRS 来使系统达到完全一致，那么我们需要将**事件溯源**和主存储保存在一个**关系型数据库**中，我们的**命令处理器**应该处理所有接收到的事件，并将它们同时存储在两个存储器中，如下图所示。



笔者认为应该通过理解业务领域来定义一致性水平。你需要了解事件中需要多大的一致性以及实现这样的一致性需要花费的成本。检查你的业务领域，考虑上述因素后你应该能做出一些决定。

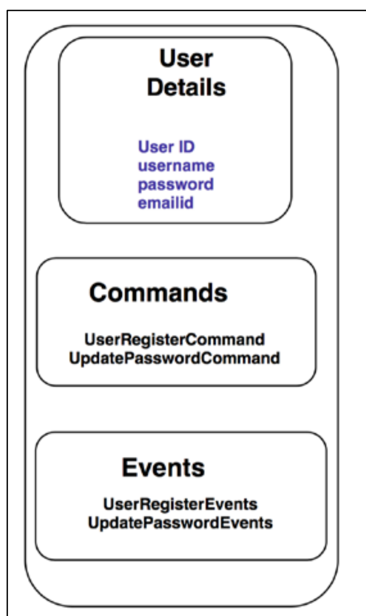
- **验证**：假如这里说的是验证客户注册表单，在该表单中验证个人信息，这很简单。但是，当必须基于唯一性进行验证时，实际问题就来了——比如具有特定用户凭证（用户名/密码）的客户。所以，要确保用户名是唯一的，当有超过 200 万的用户需要注册时，该验证至关重要。关于验证方面有如下问题：
  — 验证的数据要求是什么？
  — 从哪里检索数据进行验证？
  — 验证的可能性是多少？
  — 验证失败对业务的影响是什么？
- **并行数据更新**：这在数据一致性方面非常重要。比方说，有一个用户想要同时更新某些记录，这可能在几纳秒之内就执行完了。在这种情况下，一致性和验证检查更具挑战性，因为该用户的行为可能最终覆盖了其他用户的信息而导致系统混乱。

应对挑战

在事件溯源中解决该问题的方法之一是在事件中添加版本，处理数据更改，并确保完全验证。

解决问题

我们来看一下下图中用于事件溯源和 CQRS 的用例，思考如何为它编写代码。



问题解释

在这个例子中，我们提供了用户详细信息，如用户 ID （应该是唯一的）、用户名、密码、电子邮件 ID 等。我们必须创建两个写命令用于触发事件——UserRegistrationCommand 和 UpdatePasswordCommand，它们触发两个事件：UserRegisterEvents 和 UpdatePasswordEvents。用途是用户一旦注册，就可以根据自己的需要重置密码。

解决方案

为了解决这个问题，我们需要编写与写入命令相关的函数来接收输入并更新事件存储器。

将以下代码添加到 `commands.py` 文件中，该文件中包含写入命令相关的代码，如下所示：

```
class userregister(object):
  def __init__(self, user_id, user_name, password, emailid):
    self.user_id = user_id
    self.user_name = user_name
    self.password = password
```

```
    self.emailid = emaild

class updatepassword(object):
  def __init__(self, user_id, new_password, original_version):
    self.item_id = item_id
    self.new_password = new__password
    self.original_version = original_version
```

我们添加了与命令相关的函数，但是应该从用户的详细信息中调用。

添加一个名为 `main.py` 的新文件，调用前面命令函数。

在下面的代码中，我们通过触发事件来调用前面的代码：

```
from aggregate import Aggregate
from errors import InvalidOperationError
from events import *

class userdetails(Aggregate):
  def __init__(self, id = None, name = '"", password = "", emailid = "" ):
    Aggregate.__init__(self)
    self._apply_changes(Userdetails(id, name, password, emailid))

def userRegister(self, userdetails):
    userdetails = {1, "robin99", "xxxxxx", "robinatkevin@gmail.com"
}
    self._apply_changes(UserRegisterevent(userdetails))

  def updatePassword(self, count):
    password = ""
    self._apply_changes(UserPasswordEvent(password))
```

我们一个函数一个函数地来了解前面的代码：

```
def __init__(self, id = None, name = '"", password = "", emailid = "" ):
   Aggregate.__init__(self)
   self._apply_changes(Userdetails(id, name, password, emailid))
```

最后一行代码用默认值初始化 `self` 对象，这与其他编程语言的初始化函数类似。

接下来，定义了 userRegister 函数，收集 userdetails，然后创建事件，（UserRegisterevent（userdetails）），如下所示：

```
def userRegister(self, userdetails):
   userdetails = {1, "robin99", "xxxxxx", "robinatkevin@gmail.com"
}
   self._apply_changes(UserRegisterevent(userdetails))
```

因此，一旦用户注册，他/她有权更新配置文件的详细信息，可能是电子邮件 ID、密码、用户名或其他信息——在我们的示例中，是密码。请参考以下代码：

```
def updatePassword(self, count):
 password = ""
self._apply_changes(UserPasswordEvent(password))
```

你可以编写类似的代码来更新电子邮件 ID、用户名或其他信息。

继续，我们需要添加错误处理，如 main.py 文件中所示的，调用一个 custom module 来处理与操作有关的 errors。如果捕获到错误，将以下代码添加到 errors.py 中以传递错误：

```
class InvalidOperationError(RuntimeError):
 pass
```

正如你在 main.py 中看到的，我们调用 Aggregate 模块，你一定想知道为什么使用它。 Aggregate 模块非常重要，因为它可以跟踪应用的变化。换句话说，它强制事件将所有未注册的更改提交给事件存储器。

为了做到这一点，我们将下面的代码添加到一个名为 aggregate.py 的新文件中：

```
class Aggregate(object):
  def __init__(self):
    self.uncommitted_changes = []

  @classmethod
  def from_events(cls, events):
    aggregate = cls()
    for event in events: event.apply_changes(aggregate)
    aggregate.uncommitted_changes = []
```

```
        return aggregate

    def changes_committed(self):
        self.uncommitted_changes = []

    def _apply_changes(self, event):
        self.uncommitted_changes.append(event)
        event.apply_changes(self)
```

在 aggregate.py 中，初始化在 main.py 中调用的 self 对象，然后跟踪正在触发的事件。一段时间后，调用 main.py 中的更改来更新具有更新值和事件的 eventstore。

创建一个新的文件 events.py，其中包含需要在后端注册的命令的定义。在 events.py 中更新下面的代码片段：

```
    class UserRegisterEvent(object):
     def apply_changes(self, userdetails):
        id = userdetails.id
        name = userdetails.name
        password = userdetails.password
        emailid = userdetails.emailid

    class UserPasswordEvent(object):
     def __init__(self, password):
        self.password = password

     def apply_changes(password):
        user.password = password
```

至此，还有一个非常重要的命令处理程序，因为它决定了需要执行的操作和需要触发的事件。将以下代码添加到新文件 command_handler.py 中：

```
     from commands import *

     class UserCommandsHandler(object):
      def __init__(self, user_repository):
        self.user_repository = user_repository
```

```
    def handle(self, command):
      if command.__class__ == UserRegisterEvent:
          self.user_repository.save(commands.userRegister(command.id,
    command.name, command.password, command.emailid))
      if command.__class__ == UpdatePasswordEvent:
          with self._user_(command.password, command.original_version)
     as item:
             user.update(command.password)
  @contextmanager
    def _user(self, id, user_version):
      user = self.user_repository.find_by_id(id)
      yield user
      self.user.save(password, user_version)
```

在 `command_handler.py` 中，我们写了一个句柄函数来决定事件执行的流程。

正如你所看到的，我们调用了`@contextmanager` 模块，这个很重要。

我们再来看一个场景：假设有两个人，Bob 和 Alice，都使用相同的用户凭证。假设他们试图同时更改用户信息中的详细信息字段，如密码。现在，我们需要了解这些命令是如何得到请求的。他们的请求将首先被发送到事件存储器。如果两个用户都更新密码，那么很可能一个用户的密码将被另一个用户更新。

解决这个问题的方法是使用版本和用户模式，就像我们在上下文管理器中做的那样。以 `user_version` 作为参数，来决定用户数据的状态，一旦做了修改，就递增版本，使数据一致。

因此，在我们的例子中，如果 Bob 的修改值是先有的（当然是新版本），如果 Alice 的请求版本字段与数据库中的版本不匹配，那么 Alice 的更新请求将被拒绝。

更新完成后，我们能够重新注册和更新密码。虽然这是一个如何实现 CQRS 的示例，但你可以在其上做扩展以创建微服务。

使用 Kafka 作为事件存储

虽然你已经看到了 CQRS 的实现，但是你可能对查询 `eventstore` 的工作原理会有一

些疑问。所以下面我们将以 Kafka 作为 `eventstore` 的例子来讲解。

Kafka 通常被用作消息代理或消息队列（类似于 RabbitMQ、JMS 等）。

根据 Kafka 文档中的描述，事件溯源是一种应用程序的设计风格，应用程序的状态更改以时间顺序记录。Kafka 能够支持巨量的日志数据存储，这使得它成为以该风格构建的应用程序的优秀后端。



有关 Kafka 的更多信息，阅读以下链接中的文档：`https://kafka.apache.org/documentation/`。

Kafka 由以下几个基本部分组成。

- **生产者**：发送信息给 Kafka。
- **消费者**：消费者订阅 Kafka 中的消息流。

kafka 以如下方式工作：

- 生产者向 Kafka 中的 topic 写入消息，可能是用户。
- Kafka 中的每条消息都追加在分区的末尾。
- 分区代表事件流，而一个 topic 可以被分为多个 topic。



topic 中的分区彼此独立。

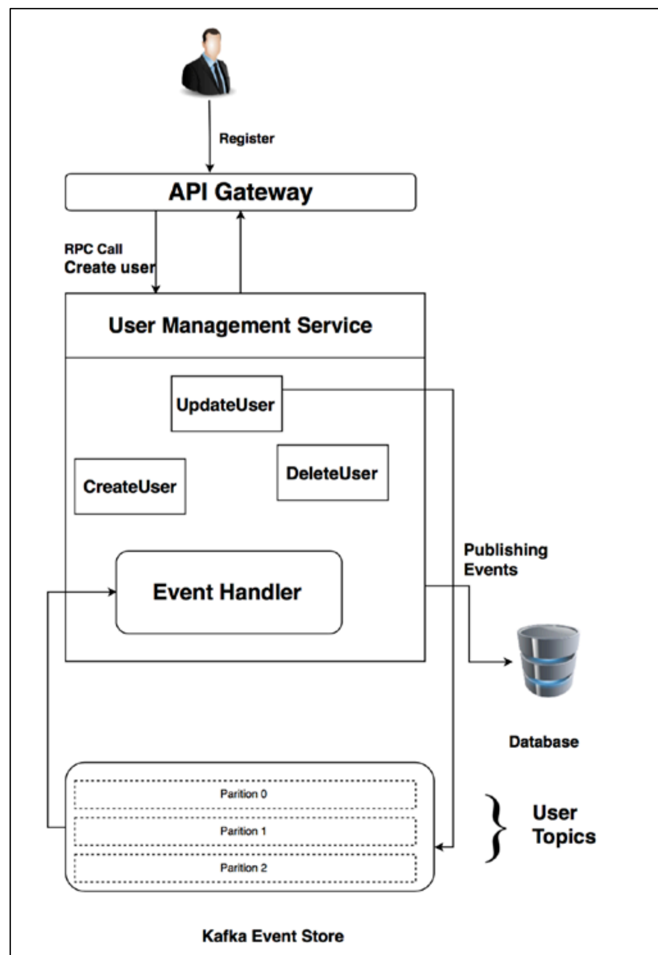
- 为了避免灾难发生，Kafka 分区被复制到多台机器上。
- 为了消费 Kafka 消息，客户端从消费者在 Kafka 中设置的偏移量处开始依次读取消息。

使用 Kafka 做事件溯源

我们来看一下客户端执行特定操作的用例，使用 Kafka 作为事件库来捕获所有正在传递的消息。在这种情况下，我们有用户管理服务，这可能是一个微服务，负责管理所有的用户请求。我们将从基于用户事件确定 Kafka 的 topic 开始，包括如下事件：

- `UserCreatedEvent`
- `UserUpdatedEvent`
- `UserDeletionEvent`
- `UserLoggedinEvent`
- `UserRoleUpdatedEvent`

在理想情况下，这些事件将由**用户管理服务**发布，所有微服务将消费这些事件。下图显示了用户请求流程。



工作原理

用户向 API 网关发出 `POST` 请求，API 网关是用户管理服务注册用户的入口点。 API 网关反过来对管理服务中的 `createUser` 方法进行 **RPC** 调用（远程过程调用）。`createUser` 端点对用户输入执行一组验证。如果输入无效，则会抛出异常，并将错误返回给 API 网关。一旦用户输入被验证，用户则被注册，并且 `UserCreatedEvent` 被触发以在 Kafka 中发布。在 Kafka 中，分区捕获事件。在我们的例子中，用户 topic 有三个分区，所以事件将基于一些定义的逻辑被发布到三个分区中的一个。这个逻辑是我们自己定义的，根据用例而定。



可以直接从 readStore（PostgreSQL 等数据库）中检索所有读取操作（如列出用户等）。

本章小结

这一章的内容有些复杂，但如果你能完全理解，对提高你的应用程序的效率和性能将很有帮助。

我们从经典架构的弊端开始介绍，然后讨论了事件溯源和 CQRS 的概念和实现。我们也研究了一个示例问题的实现，讨论了为什么这些模式是有用的，以及为什么说它们特别适用于大规模的云原生应用程序。

在下一章中，我们将深入探讨应用程序的安全性。敬请关注！

8

Web 应用的安全性

在本章中，我们将主要讨论如何保护你的应用程序免受外部威胁，这些威胁能够导致数据丢失，进而影响整体业务。

无论对于什么业务部门，Web 应用程序的安全性始终都是一个令人关注的问题。因此，我们不仅关注传统的应用程序逻辑和数据相关的安全问题，而且还关注协议和平台层面的问题。开发人员在确保遵守有关 Web 应用程序安全方面已经积累了很多最佳实践。

注意，本书面向的读者包括应用程序层面和平台层面的开发人员、系统管理人员以及希望保持其应用程序安全性的 DevOps 专业人员。

本章将涉及以下主题：

- 网络安全与应用安全
- 使用不同的方法（如 OAuth、客户端身份验证等）实施应用程序授权
- 对于开发安全的 Web 应用程序的建议

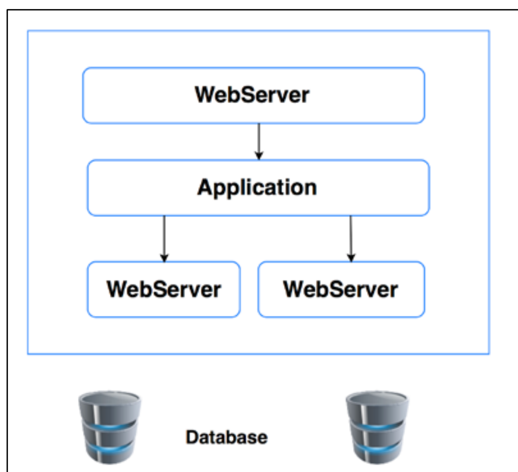
网络安全性和应用安全性

在当今，Web 应用程序的安全性取决于两个主要的层面——Web 应用程序本身和它所在的平台。这两者是浑然一体的，因为 Web 应用程序无法脱离平台而单独部署。

网络应用栈

理解平台和应用程序之间的区别非常重要，因为这对安全性有影响。典型的 Web 应用

程序将具有类似于下图所示的架构。



大多数 Web 应用程序都依赖于 Web 服务器，例如 Apache/HTTP 服务器、Rails、Nginx 等，它们根据应用程序的类型实际处理传入的请求。这些 Web 服务器跟踪传入的流量；同时它们也会验证请求，并对请求作出响应，确认用户验证是否有效。在我们的例子中将使用 Flaskacts 作为应用程序的 Web 服务器。

不同平台的应用安全性选择

如上文所述，每个 Web 应用程序都需要在某种平台上部署，然后才能向外部世界展示。应用程序平台为应用程序提供协议支持，这是通过网络进行通信所必需的。TCP 以及 HTTP 在很大程度上都是在应用程序级别处理的。

在软件架构的网络栈中，应用程序平台中有两个截然不同的层次，其中包含了适用于 Web 应用程序攻击的协议。这些层次如下：

- 传输
- 应用程序

我们来看看分层的细节。

传输协议

在**开放系统互连**模型（**OSI** 模型）中，传输层通常被称为第四层。因为其可靠性，Web

应用程序使用 TCP 作为传输协议。

在 **TCP**（**传输控制协议**）中，每个数据包都受到密切监控，并且内置错误恢复机制，这在出现通信故障时非常有用。这些机制可能被利用来攻击 Web 应用程序。

最常见的攻击是 **SYN flood** 攻击，这是一个确认攻击的 TCP 请求。SYN Flood 攻击会严重影响应用，其使用空闲会话与应用服务器建立连接，一直持续请求，直到服务器资源耗尽，不能再处理更多的请求。

为了避免这种类型的攻击，系统管理员（开发者无法控制）在评估客户影响后，应该建立与超时和空闲行为有关的配置。这种类型的攻击的另一个例子是 **Smurf 攻击**（参阅此链接了解更多详情：`https://en.wikipedia.org/wiki/Smurf_ attack`）。

安全传输协议

在 OSI 网络模型中，在第五层也有一些协议，可以使网络更安全可靠——SSL/TLS。然而，这一层也有一些漏洞（例如 2014 年针对 SSL 协议的 Heartbleed 攻击和 2009 年发生的 TLS 中间人重新谈判攻击）。

应用程序协议

在 OSI 网络模型的第七层（最上层）上，驻留着实际的应用程序，它使用 HTTP 协议进行通信，这是大多数应用程序攻击发生的地方。

HTTP（超文本传输协议）主要包括两个组件。

- **元数据**：HTTP header 中包含元数据，这对于应用程序及平台都很重要。header 中可能包括 cookie、content-type、状态、连接等。
- **行为**：其定义了客户端和服务器之间的行为。定义在 HTTP 客户端（例如浏览器）和服务器之间应该如何交换消息。

这里的主要问题是，应用程序通常不会内置识别可疑行为的功能。

例如，客户端通过网络访问 Web 应用程序，网络可能受到**拒绝服务**（**DoS**）攻击。在这种攻击中，客户端有意识地以比正常情况下更慢的速率接收数据，这样就可以使打开的连接保持的时间更长。因此，Web 服务器的队列开始被塞满，消耗更多的资源。如果所有的资源都被用于保持开放连接，服务器很可能无法响应。

应用程序逻辑中的安全威胁

在本节中，我们将介绍对用户进行身份验证的不同方法，确保我们的应用程序由真实实体访问。

Web 应用程序安全替代方案

为了确保我们的应用程序免受外部威胁，这里列举了几种替代方案。通常，我们的应用程序没有识别可疑活动的能力。因此，有了以下这些重要的安全措施：

- 基于 HTTP 的身份验证
- OAuth/OpenID
- Windows 身份验证

基于 HTTP 的身份验证

简单的**用户名**和**密码**被哈希散列后由客户端发送到 Web 服务器，就像下图所示的 Web 应用程序的设置。

该图是我们在第 6 章中创建的用户界面。后端服务（微服务）和用户数据库进行身份验证，身份信息存储在 MongoDB 数据库服务器中。此外，验证用户登录主页时，从 MongoDB 集合中读取用户数据，用户通过身份验证后进入应用程序。以下是调用 API 的代码片段：

```
@app.route('/login', methods=['POST'])
def do_admin_login():
 users = mongo.db.users
 api_list=[]
 login_user = users.find({'username': request.form['username']})
 for i in login_user:
   api_list.append(i)
  print (api_list)
  if api_list != []:
    #print (api_list[0]['password'].decode('utf-8'),
     bcrypt.hashpw(request.form['password'].encode('utf-8'),
     api_list[0]['password']).decode('utf-8'))
  if api_list[0]['password'].decode('utf-8') ==
     bcrypt.hashpw(request.form['password'].encode('utf-8'),
     api_list[0]['password']).decode('utf-8'):
       session['logged_in'] = api_list[0]['username']
       return redirect(url_for('index'))
       return 'Invalide username/password!'
  else:
    flash("Invalid Authentication")
 return 'Invalid User!'
```

这是在应用程序级别设置安全性，这种方式可以保障应用程序数据库安全。

OAuth/OpenID

OAuth 是一个开放的授权标准，在允许用户使用第三方证书（通常是电子邮件 ID）进行身份验证的网站中非常常见。

下面列出了 OAuth 比其他安全措施更好的几个关键特性：

- 它与任何操作系统（操作系统）或部署都没有关系。
- 简单易用。
- 更可靠，并提供高性能。
- 专门设计用于需要集中认证方式的分布式系统。
- 这是一款免费使用的基于开放源码的身份提供商服务器的软件。
- 支持基于云的身份提供者，如 Google、Auth0、LinkedIn 等。
- 也称为 SSO（单点登录或基于令牌的认证）。

- 设置管理员账户。

如果没有服务授予 JWT（JSON Web Token，URL 安全的 JSON 格式，用于表达可在各方之间传输的声明），OAuth 将无法正常工作。你可以通过 `https://jwt.io/introduction/` 了解有关 JWT 的更多信息。

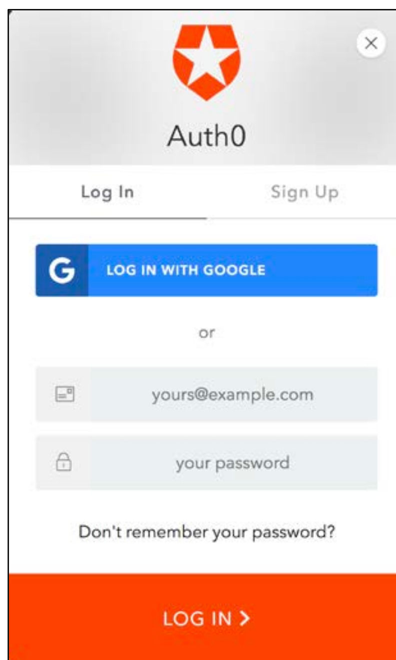
身份提供者负责根据第三方提供的授权来对 Web 应用程序进行身份验证。

你可以根据自己的偏好选择任何身份提供商，因为他们的性质相似，但是在功能上会有所不同。在本章中，我们将介绍如何使用 Google Web 应用程序（来自 Google 的开发者 API）和 Auth0 第三方应用程序进行身份验证。

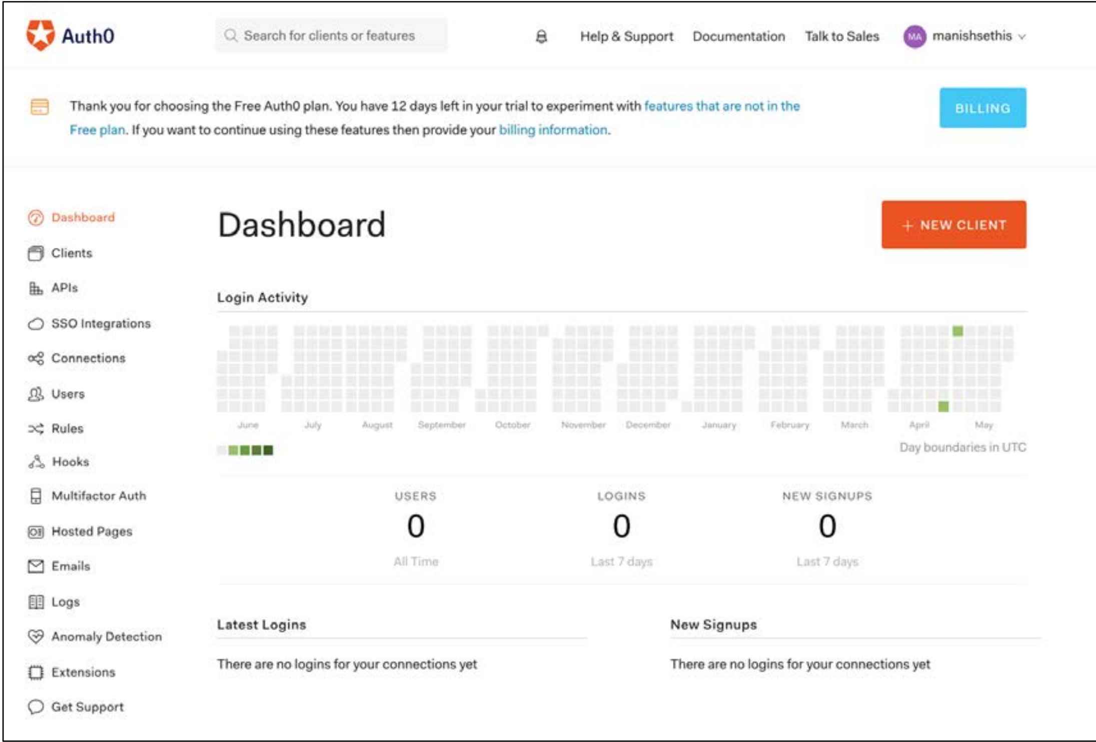
创建 Auth0 账户

在本节中，我们将在 Google 开发者工具中建立一个用于验证的账户，并在名为 **Auth0**（auth0.com）的第三方免费应用程序中建立一个账户。

我们在 Auth0（auth0.com）中设置账户，其中唯一的要求就是需要使用电子邮件 ID 来注册，如下图所示。



注册了 Auth0 账户后，你将看到如下图所示的屏幕。



这是仪表板，我们可以将登录活动看作登录到应用程序的用户。它还显示用户的登录尝试，并保存用户活动的日志。简而言之，仪表板可以让你深入了解应用程序的用户活动。

现在我们为应用程序添加一个新的客户端，单击+ **NEW CLIENT** 按钮来创建。单击 **+NEW CLIENT** 按钮后，将出现如下图所示的画面。

Create Client

Name

My App

You can change the client name later in the client settings.

Choose a client type

Native
Mobile or Desktop, apps that run natively in a device.
eg: iOS SDK

Single Page Web Applications
A JavaScript front-end app that uses an API.
eg: AngularJS + NodeJS

Regular Web Applications
Traditional web app (with refresh).
eg: Java ASP.NET

Non Interactive Clients
CLI, Daemons or Services running on your backend.
eg: Shell Script

CREATE

可以看到，需要为客户端提供一个名称（通常，名称应该与应用程序相关）。另外，还需要选择应用程序的类别。回到我们的案例，我们已经给出了应用程序的名称，并选择了第二个选项，即 **Single Page Web Applications**（**单页 Web 应用程序**），因为我们要使用它里面的技术。或者，也可以选择 **Regular Web Applications**（**普通 Web 应用程序**），它工作得很好。这些选项用于区分应用程序的类型，因为我们很可能在一个账户下开发数百个应用程序。

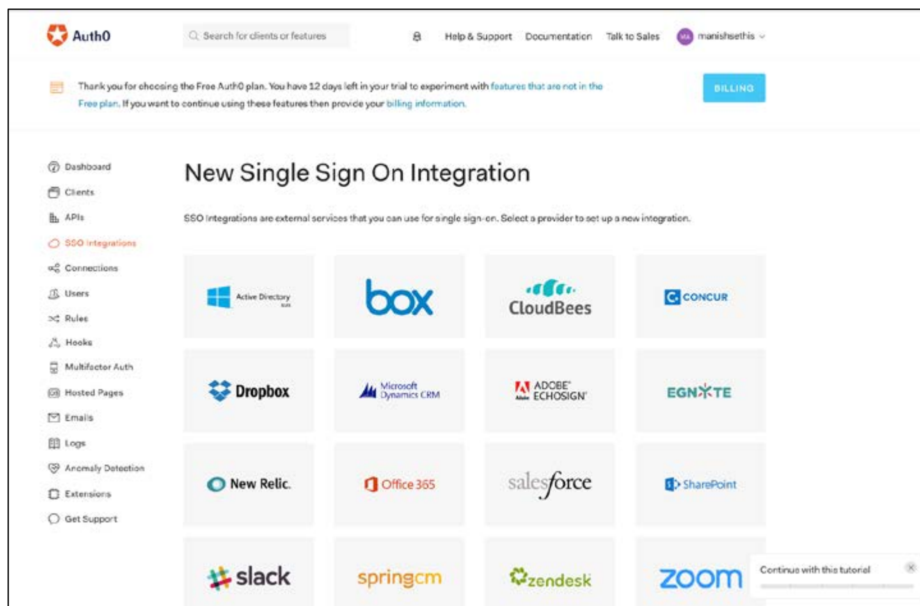
单击 **CREATE** 按钮继续创建客户端。创建完成后，你将看到如下图所示的画面。

在该图中，有很多设置是自动生成的，我们需要将它们与我们的 Web 应用程序集成。定义了如下这些项。

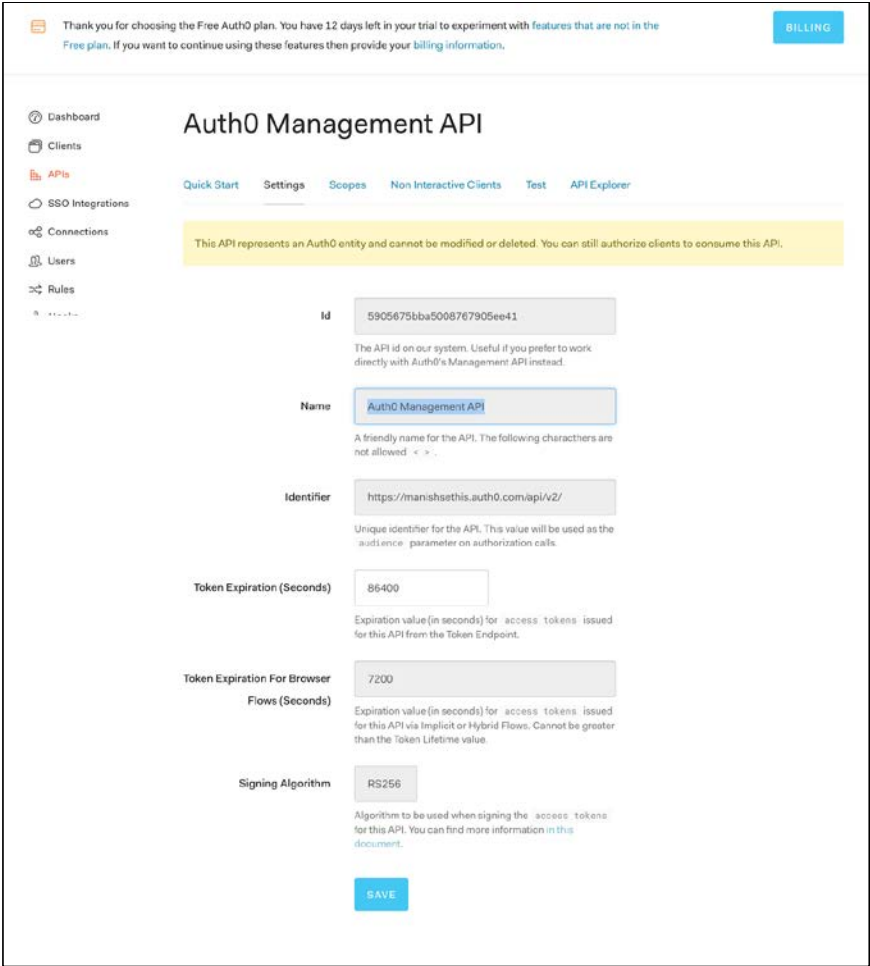
- **Client ID（客户端 ID）**：这是分配给特定应用程序域的唯一 ID。
- **Domain（领域）**：这与认证服务器类似，将在应用程序登录时调用。
- **Client Secret（客户端密码）**：这是一个密钥，应保持安全，不要与任何人共享，否则可能导致安全漏洞。
- **Client Type（客户端类型）**：这定义了应用程序的类型。
- **Allowed Callback URLs（允许回调的 URL）**：指定用户认证后允许的回调 URL，例如 `http://localhost: 5000/callback`。
- **Allowed Logout URLs（允许注销的 URL）**：这个定义了在用户注销时可以点击的 URL，比如 `http://localhost5000/logout`。
- **Token Endpoint Authentication Method（令牌端点验证方法）**：这定义了验证的方法，可以是 none、post 或 basic。

用来管理应用程序 Auth0 账户的还有如下这些项。

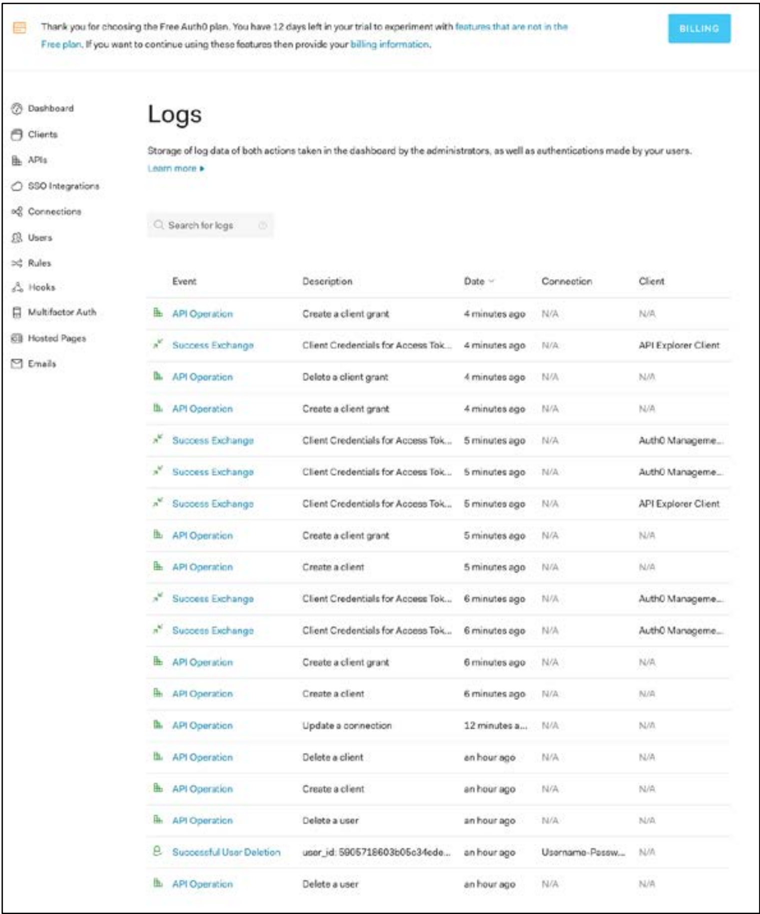
- **SSO Integrations（SSO 集成）**：可以使用其他第三方应用程序（如 Slack、Salesforce、Zoom 等）设置 SSO 登录，如下图所示。



- **Connections（连接）**：它定义了你想为应用程序定义的认证类型，如数据库（用户名和密码数据库）、社交（与社交媒体网站如 Google、LinkedIn 等现有账户的整合）、Enterprise（企业级应用程序，如 AD、Google Apps 等）或无密码（通过短信、电子邮件等）。默认情况下，启用了用户名和密码认证。
- **API**：在这里管理应用程序的 **Auth0 Management API** 并对其进行测试，如下图所示。



- **Logs（日志）**：将跟踪你在 Auth0 账户上的活动，这对于调试以及识别可疑活动的威胁非常有用。参阅下图以查找有关日志的更多信息。



这些是管理 Auth0 账户最重要的功能，可以帮助你高效地管理 Web 应用程序的安全性。

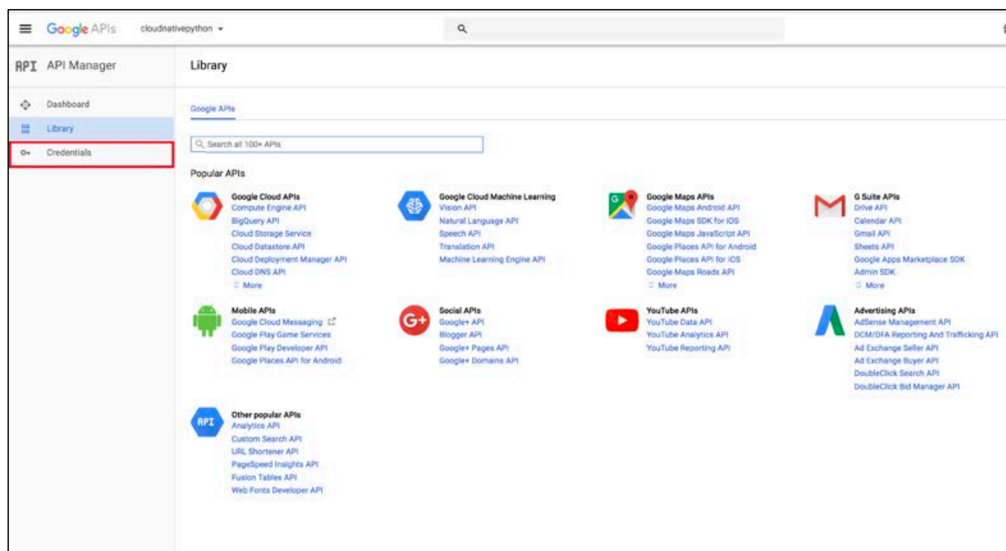
现在，我们的 Auth0 管理员账户已经建立好了，并且已经准备好与我们的 Web 应用程序集成。

创建 Google API 账户

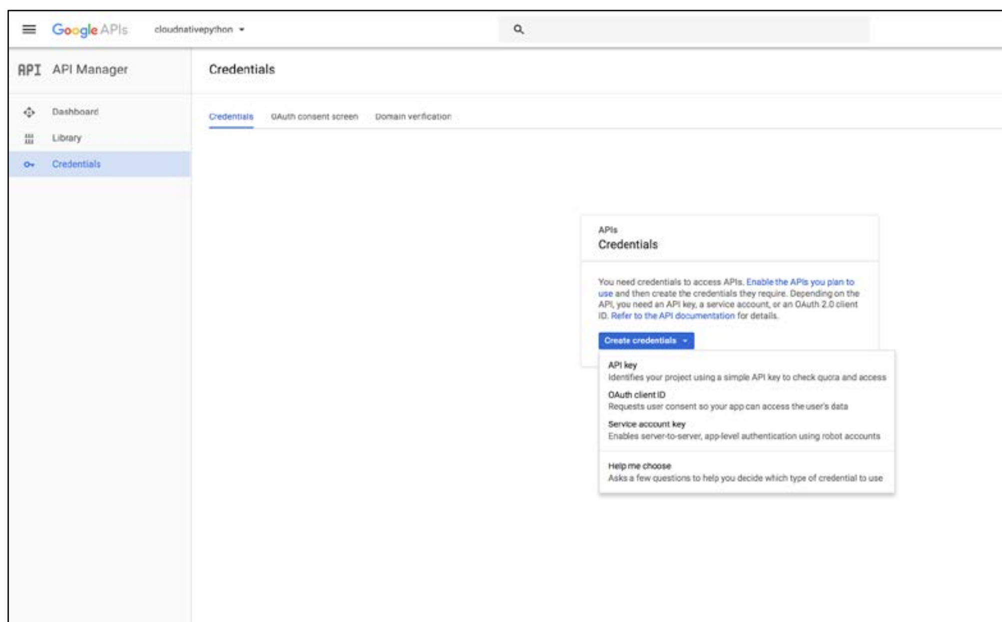
Google API 使用 OAuth 2.0 协议进行身份验证和授权。Google 支持常见的 OAuth 2.0 场景，例如 Web 服务器、已安装的应用程序和客户端应用程序。

使用你的 Google 账户登录 Google API 控制台（`https://console.developers.google.com`）以获取 OAuth 客户端凭证，例如**客户端 ID**、**Client Secret** 和其他凭证。需

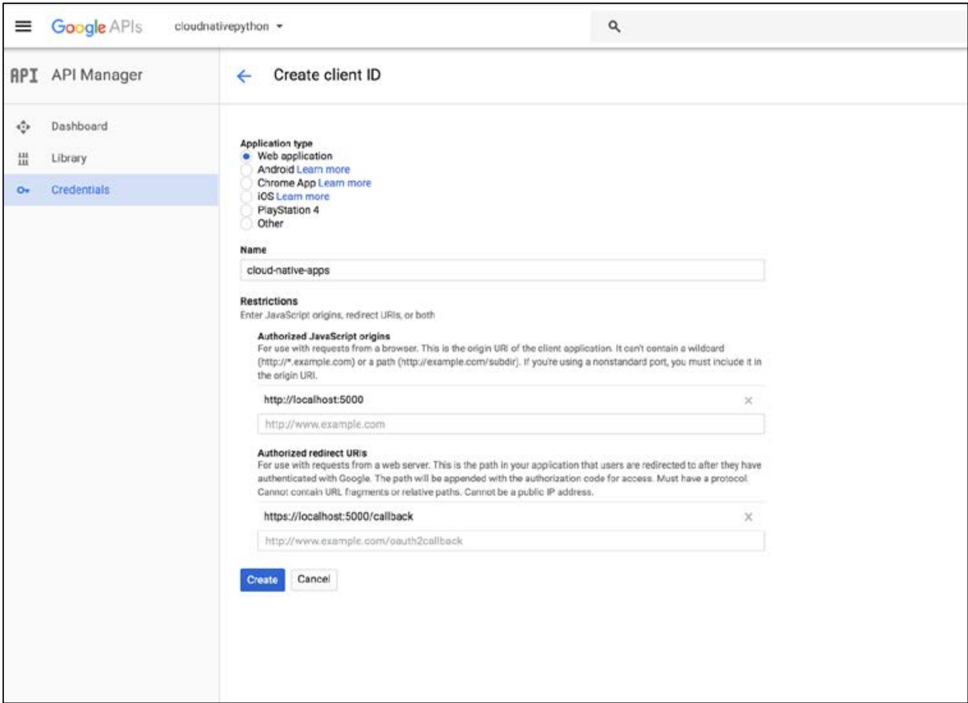
要将这些凭证与应用程序集成。登录后，你将看到如下图所示的画面。



该图显示了针对不同 Google 产品的 Google API。现在，单击左侧面板中的 **Credentials**（**凭证**）选项，导航到下一个页面，如下图所示。



现在，单击 **Create credentials（创建凭证）**选项，然后单击 **OAuth client ID（OAuth 客户端 ID）**选项以从 API 管理器生成客户端凭证，如下图所示。



现在我们需要提供我们的应用程序的一些信息。你还必须记住在创建 OAuth 账户时提供的这些信息。准备好并填写了必填字段后，单击 **Create（创建）**按钮生成凭证。

客户端 ID 创建后，你将看到如下图所示的画面，其中包含与客户端 ID（**证书**）相关的信息。

记住，永远不要与任何人共享客户端 ID 的细节。如果你这样做了，请立即重置。现在我们的 Google API 账户已准备好与 Web 应用程序集成。

在 Web 应用程序中集成 Auth0 账户

为了将 Auth0 账户与我们的应用程序集成，我们需要为我们的回调创建一个新的路由。此路由将在 Auth0 账户用户身份验证后建立会话。所以，将下面的代码添加到 `app.py` 文件中：

```
@app.route('/callback')
def callback_handling():
  code = request.args.get('code')
  get_token = GetToken('manishsethis.auth0.com')
  auth0_users = Users('manishsethis.auth0.com')
  token = get_token.authorization_code(os.environ['CLIENT_ID'],
                                  os.environ['CLIENT_SECRET'],
  code, 'http://localhost:5000/callback')
  user_info = auth0_users.userinfo(token['access_token'])
```

```
  session['profile'] = json.loads(user_info)
return redirect('/dashboard')
```

如你看到的，这里使用了从 Auth0 账户控制台获取的客户端凭证。这是我们在创建客户端时生成的凭证。

现在我们添加用户在通过身份验证后被重定向到的路由/仪表板：

```
@app.route("/dashboard")
def dashboard():
  return render_template('index.html', user=session['profile'])
```

这个路由只是调用 `index.html`，并将会话的详细信息作为参数传递给 `index.html`。

现在我们必须修改 `index.html` 来通过 Auth0 触发验证。有两种触发方式。第一种是将 Auth0 域作为登录页面，这意味着只要用户单击 URL（`http://localhost5000`），他们就会被重定向到 Auth0 账户的登录页面。另一种方法是通过提供一个触发按钮来手动触发它。

本章，我们将使用手动触发器，其中 Auth0 账户可以用作登录到应用程序的替代方案。

将下面的代码添加到 `login.html` 中。这段代码会在登录页面上显示一个按钮，如果你单击那个按钮，将触发 Auth0 用户注册页面：

```
<center><button onclick="lock.show();">Login using Auth0</button>
  </center>
<script src="https://cdn.auth0.com/js/lock/10.14/lock.min.js">
  </script>
<script>
 var lock = new Auth0Lock(os.environ['CLIENT_ID'],
  'manishsethis.auth0.com', {
   auth: {
    redirectUrl: 'http://localhost:5000/callback',
    responseType: 'code',
    params: {
      scope: 'openid email' // Learn about scopes:
      https://auth0.com/docs/scopes
     }
    }
```

```
  });
</script>
```

在测试应用程序之前还有一件事需要注意——如何让应用程序知道会话的细节。

由于 `index.html` 采用会话值并在主页上显示它们，所以它将被用来管理来自用户的推文。

所以，更新 `index.html` 的 body 标签为如下所示：

```
<h1></h1>
<div align="right"> Welcome {{ user['given_name'] }}</div>
<br>
<div id="react"></div>
```

这段代码在用户界面上显示用户的全名。接下来，需要更新 `localStorage` 会话详细信息，如下所示：

```
<script>
  // Check browser support
  if (typeof(Storage) !== "undefined") {
  // Store
  localStorage.setItem("sessionid","{{ user['emailid'] }}" );
  // Retrieve
  document.getElementById("react").innerHTML =
  localStorage.getItem("sessionid");
  } else {
    document.getElementById("react").innerHTML = "Sorry, your
    browser does not support Web Storage...";
  }
</script>
```

现在差不多完成了。希望你还记得在微服务 API 中，向特定用户发送推文时，我们已经设置了身份验证检查。现在需要删除这些检查，改用 Auth0 进行身份验证。

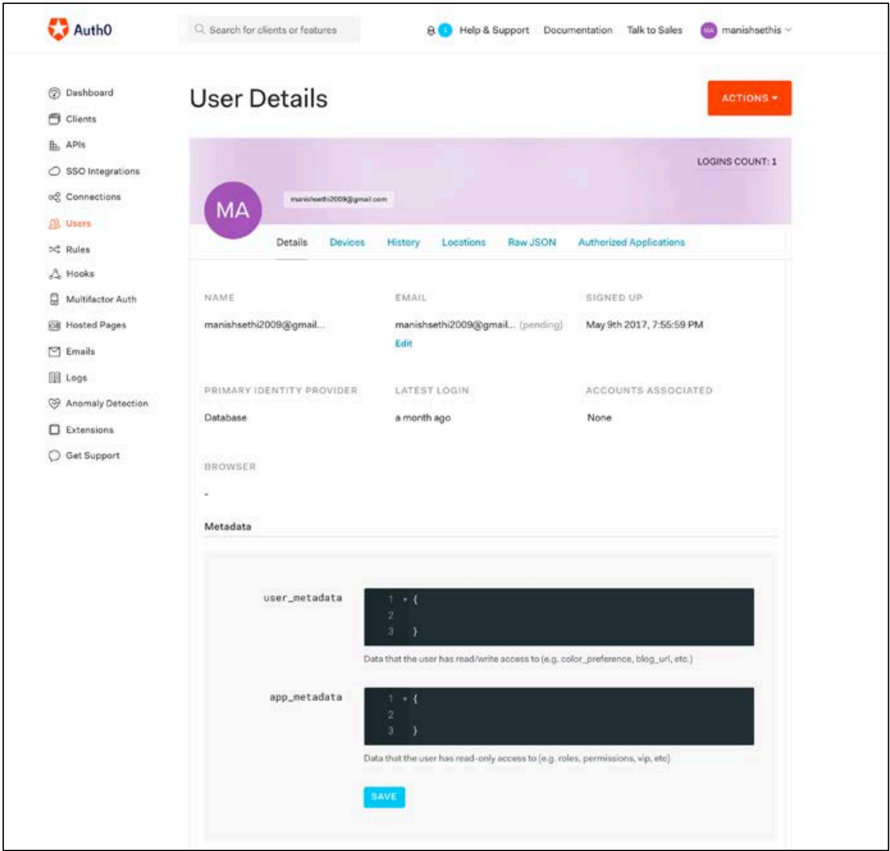
真棒！运行应用程序，看看是否可以在 `http://localhost:5000/`上看到如下图所示的画面。

接下来，单击 **Login using Auth0**（**登录 Auth0**）按钮打开 Auth0 登录/注册面板。

提供所需的信息，然后单击 **Sign up now**（**立即注册**），将在 Auth0 账户中注册。请记住，在这种情况下，你看不到通过电子邮件直接登录的任何情况，因为我们使用用户名和密码认证。如果你想通过电子邮件直接注册，则需要在社交连接部分启用 google-OAuth2 方式来扩展程序。一旦启用了它，你将看到如下图所示的注册页面。

你成功注册后，将被重定向到主页，在那里你可以发送推文。如果你看到下图所示的界面，则意味着它已正常工作。

这里需要注意的一点是，每次注册时，会在 Auth0 账户中使用 **User Details** 创建用户，如下图所示。



真棒！现在，你的应用程序与 Auth0 账户集成在一起了，你可以密切关注使用你应用程序的用户。

将 Google API 与 Web 应用程序集成

将 Google API 与 Web 应用程序集成非常类似于 Auth0 集成。你需要按照下面的步骤来集成 Google API。

1. **收集 OAuth 凭证**：正如在 Google API 客户端设置中所述，我们已经生成了客户端凭证。我们需要捕捉诸如客户端 ID、客户端秘钥等的细节。

2. 从 Google 授权服务器获取访问令牌：在你的应用程序用户可以登录并访问私人数据之前，需要生成一个由 Google 提供的身份验证令牌，该令牌充当用户的身份验证者。单个访问令牌可以授予对多个 API 的不同程度的访问权限。范围参数包含有关用户可以访问的范围信息，即用户可以从哪个 API 查看数据。令牌的请求方式取决于你的应用程序开发的方式。

3. 将令牌保存到 API：一旦应用程序接收到令牌，就会将该令牌发送到 Google API HTTP 授权 header。如前所述，该令牌有权根据定义的作用域参数对特定的一组 API 执行操作。

4. 刷新令牌：最好的做法是在一段时间后刷新令牌以避免任何安全漏洞。

5. 令牌到期：在一段时间之后进行令牌到期写入是一个很好的做法，这可以保障应用程序的安全，强烈建议你这么做。

由于我们开发的是基于 Python 的应用程序，因此你可以在以下链接中关注包含基于 Google-API 令牌的身份验证实施信息的文档 URL：

```
https://developers.google.com/api-client-library/python/guide/aaa_oauth
```

一旦用户通过身份验证，并开始使用应用程序，你就可以在 **API Manager**（`https://console.developers.google.com/apis/`）上监视用户登录活动，如下图所示。



使用 Google 设置身份验证有点困难，因为整个过程需要监督。这就是为什么开发人员喜欢使用 Auth0 之类的工具的原因，因为可以直接与 Google 进行整合。

Windows 权限认证

从历史上看，即使内部网和企业网站部署在内部或私有云中，也首选此选项。但是，由于多种原因，这不适合作为云原生安全选项。



有关 Windows 身份验证的更多信息，请参考链接 https://en.wikipedia.org/wiki/Integrated_Windows_Authentication 中的介绍。为了便于理解，我们已经展示了这些安全性方法，但身份验证方法保持不变。

开发安全的 Web 应用程序建议

随着**万维网（WWW）**Web 应用程序的增加，人们对应用程序安全性的担忧也在加重。现在，我们脑海中出现的第一个问题就是为什么需要安全的应用程序——对此的回答是非常明显的。但是它的基本原理是什么呢？以下是我们应该牢记的原则：

- 如果黑客熟悉创建应用程序所使用的语言，就可以轻松攻破应用程序。这就是为什么我们使用诸如 CORS 之类的技术来保护代码的原因。
- 访问应用程序及其数据的权限应该授给组织中非常有限的人员。
- 使用身份验证授权可以保护你的应用程序免受 WWW 以及私人网络的影响。

所有这些因素，或者说原则，都促使我们要创建安全的应用程序。

本章小结

在本章中，我们通过定义不同应用程序堆栈上的安全性，并根据你的偏好和应用程序要求，将不同安全措施集成到我们的应用程序中。

到目前为止，我们一直在讨论应用程序的构建。但是此后，我们将专注于构建一个使用 DevOps 工具将应用程序从开发阶段转移到生产阶段的平台。所以，事情会变得更加有趣。请继续阅读更多章节。

9

持续交付

在上一章中，我们构建了应用程序并已准备好将其部署到云环境上。现在我们的应用程序很稳定并准备首次发布，此时我们需要考虑平台（即云平台）以及可以帮助我们将应用程序迁移到生产环境的工具。

本章将讨论以下主要内容：

- 持续集成和持续交付简介
- 理解基于 Jenkins 的持续集成

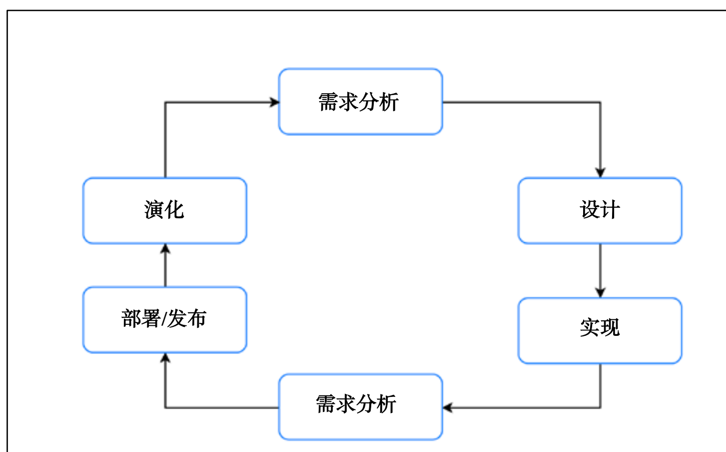
持续集成与持续交付的变迁

现在很多人都在谈论 **CI**（**持续集成**）和 **CD**（**持续交付**），在参考多个技术人员的观点之后，笔者发现大家对 CI 和 CD 有着不同的理解，并且还有些混乱。下面我们来好好了解下它们。

为了理解持续集成，需要首先了解 **SDLC**（**系统开发生命周期**）和**敏捷软件开发**的背景，这可以在构建和发布过程中为你提供帮助。

理解 SDLC

SDLC 是规划、开发、测试和部署软件的过程。这个过程由一系列的阶段组成，上一阶段的输出作为下一阶段的输入。下图描述了 SDLC。



我们来详细了解下每个阶段。

- **需求分析**：这是问题分析的初始阶段，业务分析员进行需求分析，了解业务需求。可以是组织内部的需求，也可以是外部客户的需求。需求涉及问题的范围，可以是改进系统或建立新系统、成本分析和项目目标等。
- **设计**：在这个阶段，准备和批准软件解决方案功能实现的设计，包括流程图、文档、布局等。
- **实现**：在这个阶段，执行基于设计的实际实施过程。通常，开发人员根据设计阶段中定义的目标编写代码。
- **测试**：在这个阶段，QA（质量保证）小组将在不同场景下测试开发代码。对每个模块都进行单元测试及集成测试。在测试失败的情况下，开发人员需要了解 bug，然后修复它。
- **部署/发布**：在这个阶段，测试过的功能将被发布到生产环境供客户评估。
- **演化**：这一阶段让客户对开发、测试和发布的升级版进行评估。

敏捷开发流程

敏捷软件开发过程是传统软件开发的一种替代方案。它更像是一个以最少的 bug 频繁有效地发布软件到生产环境的流程。

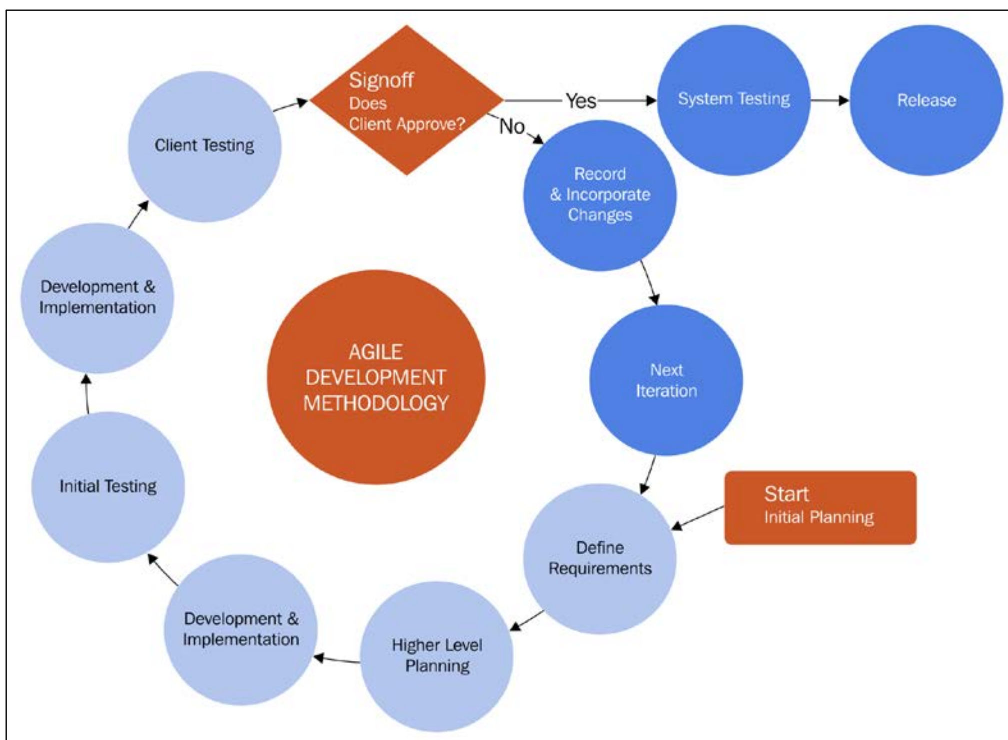
敏捷过程基于以下原则：

- 在每个阶段都持续交付软件升级版和客户反馈。

- 在开发周期的任何阶段都欢迎进一步的改进。
- 应该时常发布稳定版本（在几周内）。
- 业务团队和开发团队之间应该持续沟通。
- 持续改进技术和设计。
- 可运行的软件是进步的主要标准。
- 不断适应环境的变化。

敏捷开发运作方式

在敏捷软件开的发过程中，整个系统分为不同的阶段，所有模块和功能都是迭代交付的，跨职能团队需要做规划、单元测试、设计、需求分析、编码等各个方面的工作。因此，每个团队成员都要参与到这个过程中，没有一个人坐着闲着。而在传统的 SDLC 中，当软件处于开发阶段时，剩下的团队成员要么闲着，要么没有充分使用。所有这一切使得敏捷开发比传统模式更具优势。下图显示了敏捷开发的工作流程。



在该图中，你不会发现任何需求分析和设计阶段，因为这些阶段是在高层计划中积累的。

以下是敏捷开发过程中的事件顺序：

1. 从最初的规划开始，给出软件功能的细节，然后在高层规划中定义目标。

2. 一旦设定了目标，开发人员就开始编写所需的功能。软件升级准备就绪后，测试团队（QA）将开始执行单元测试和集成测试。

3. 如果发现任何 bug，则立即修复，然后将代码交付给客户端测试（即在 stage 环境或者说预生产环境中）。在这个阶段，代码还没有发布。

4. 如果代码通过了所有基于客户端的测试（可能是基于 UI 的测试），那么代码将被推向生产；否则，再次遍历上述周期。

现在我们已经了解了敏捷的工作流程，下面我们来看看它比传统的 SDLC 有哪些优势，具体如下：

- 在敏捷开发过程中，每个功能都可以被频繁快速地开发和演示。这里的原则是，在软件部署的一个星期内，在新开发的功能中没有发现 bug。这确保了客户对附加功能达到满意。
- 没有单独的开发、测试或其他团队。一个团队由 8 ~ 10 个成员组成（根据要求），每个成员都可以做任何事情。
- 敏捷可以促进团队合作。
- 需要的文档最少。
- 敏捷最适合并行功能的开发。

因为有这些优势，现在很多公司都开始在他们的软件开发中采用敏捷 SDLC。

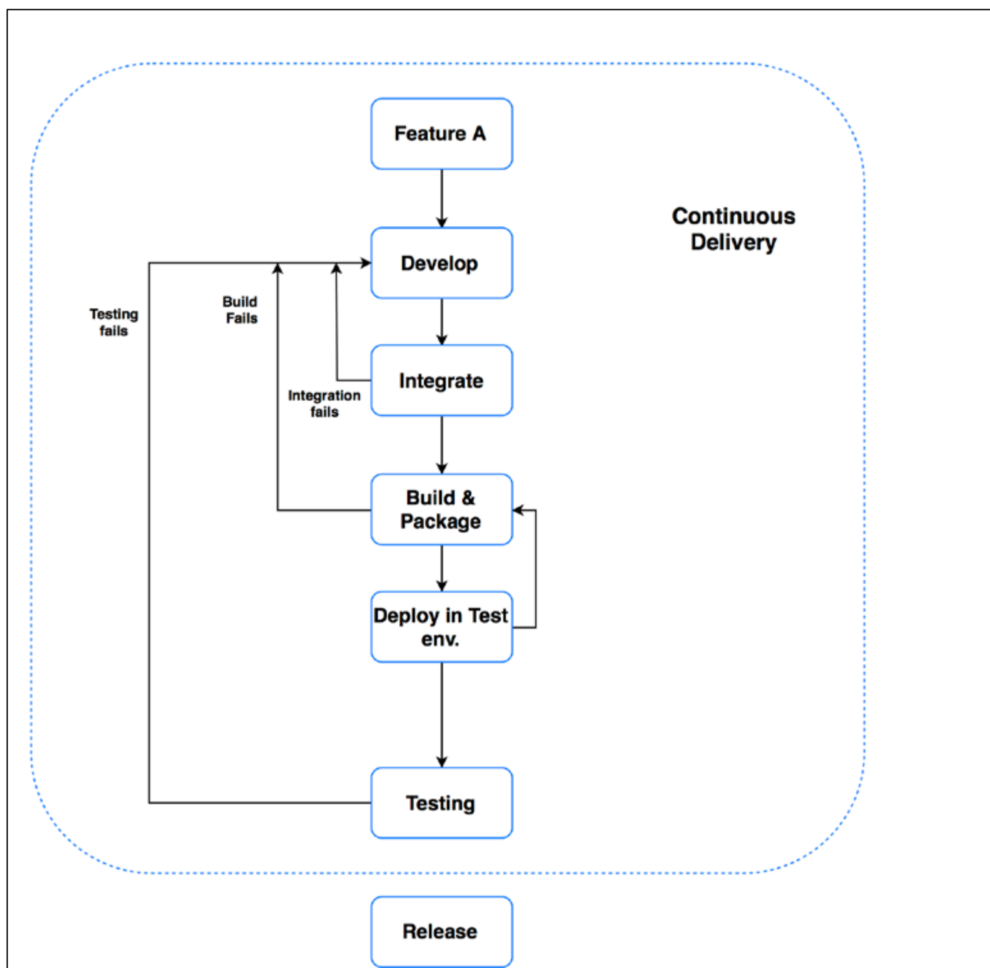
探讨了软件开发的方法后，下面我们就来看看敏捷过程中的一个非常关键的方面，那就是持续集成，其使得我们的开发工作更加容易。

持续集成

持续集成是将代码合并到主线代码库的过程。简而言之，持续集成可以帮助开发人员

在初始阶段测试新代码，在开发和生成测试结果的同时进行频繁构建，如果一切正常，则将代码合并到主线代码中。

可以通过下面的图来理解这个过程，该图描述了在 SDLC 期间发生的事件。



在持续集成过程中会出现以下问题：

- 在集成之前失败
- 集成失败
- 构建失败（在集成之后）

为了解决这些问题，开发人员需要修改代码，再次重复整个集成过程，直到成功部署。

Jenkins 持续集成工具

Jenkins 是一个开源工具，用于执行持续集成和构建自动化。它与其他任何持续集成工具（如 CirclCI）一样，在开发阶段尽可能早地测试代码。

在 Jenkins 中，你可以定义一组指令，以在不同的应用程序环境（开发、预生产阶段等）上部署应用程序。

在 Jenkins 中建立 job（一般是一个项目）之前，我们先来了解一下 Jenkins 插件。我们先来设置 Jenkins 并按照我们的要求进行配置。

安装 Jenkins

无论是在 Linux（Debian、Red Hat 等）、Windows 还是 Mac OS 下，Jenkins 的安装都很简单。

前提条件

确保你的 Ubuntu 系统上安装了 Java 8。如果没有安装，可以按照以下链接中的说明安装：

```
https://medium.com/appliedcode/how-to-install-java-8-jdk-8u45-on-ubuntu-
linuxmint-via-ppa-1115d64ae325
```

在 Debian 系统（Ubuntu）上安装 Jenkins

按照下面的步骤在基于 Debian 的系统上安装 Jenkins。

1. 执行以下命令，将 Jenkins 键添加到 APT 软件包列表来安装：

```
$ wget -q -O - https://pkg.jenkins.io/debian/jenkins-ci.org.key | sudo
apt-key add –
```

2. 接下来，使用需要连接的服务器来更新源文件以验证密钥，如下所示：

```
$ sudo sh -c 'echo deb http://pkg.jenkins.io/debian-stable binary/ >
/etc/apt/sources.list.d/jenkins.list'
```

3. 源列表文件被更新后，在终端上执行以下命令来更新 APT 存储库：

```
$ sudo apt-get update -y
```

4. 现在我们准备在 Ubuntu 上安装 Jenkins 了。使用以下命令来安装：

```
$ sudo apt-get install jenkins  -y
```

5. 安装完成了，请记住 Jenkins 默认运行在 8080 端口上。但是如果你想让它运行在其他的端口上，需要更新 Jenkins 配置文件（`/etc/default/jenkins`）中的以下行：

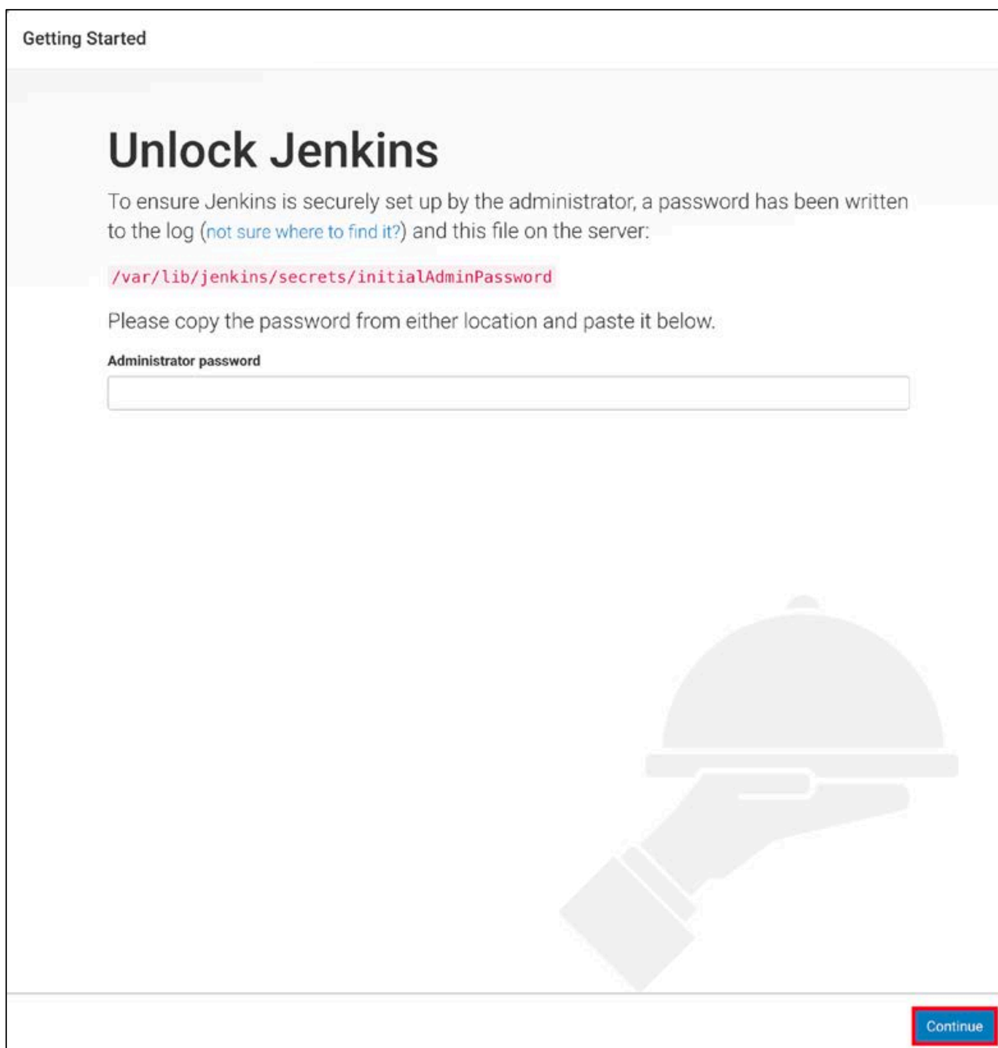
```
HTTP_PORT=8080
```

6. 接下来，通过 URL 检查 Jenkins 的 GUI：

- 如果是在本地安装，请访问 `http://localhost:8080/`
- 如果是在远程机器上安装，请访问 `http://ip-address:8080`

请记住，我们当前安装的是 Jenkins 2.6.1 版本，之前及后续步骤对于 Jenkins 2.x.x 版本都适用。

如果你看到下图所示的界面，表示安装成功。



从该图可以看出，在 Jenkins 安装的系统中的一个目录下保存着默认的密码。

这证明 Jenkins 已经成功安装。

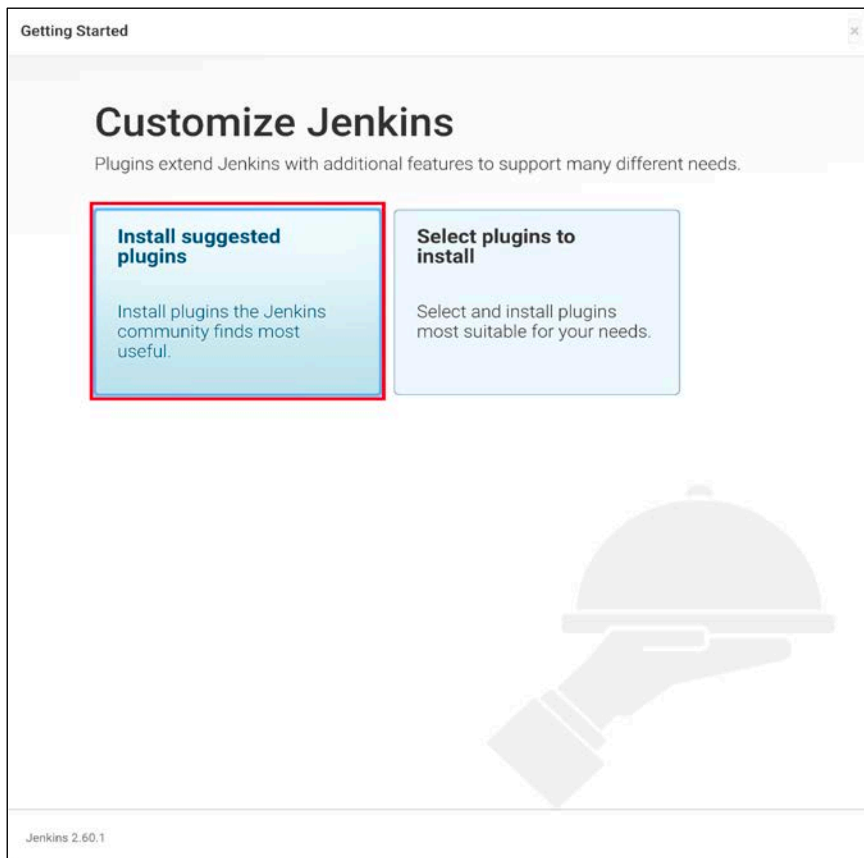


在 Windows 上安装

在 Windows 上安装 Jenkins 相当简单。通常，Jenkins 在 Windows 上不是以服务的方式运行。但是如果你想使用服务模式（可选的），请参考下面 URL 中的 Jenkins 完全安装文档：`https://wiki.Jenkins-ci.org/display/JENKINS/Installing+Jenkins+as+a+Windows+service#InstallingJenkinsasaWindowsservice-Insta`

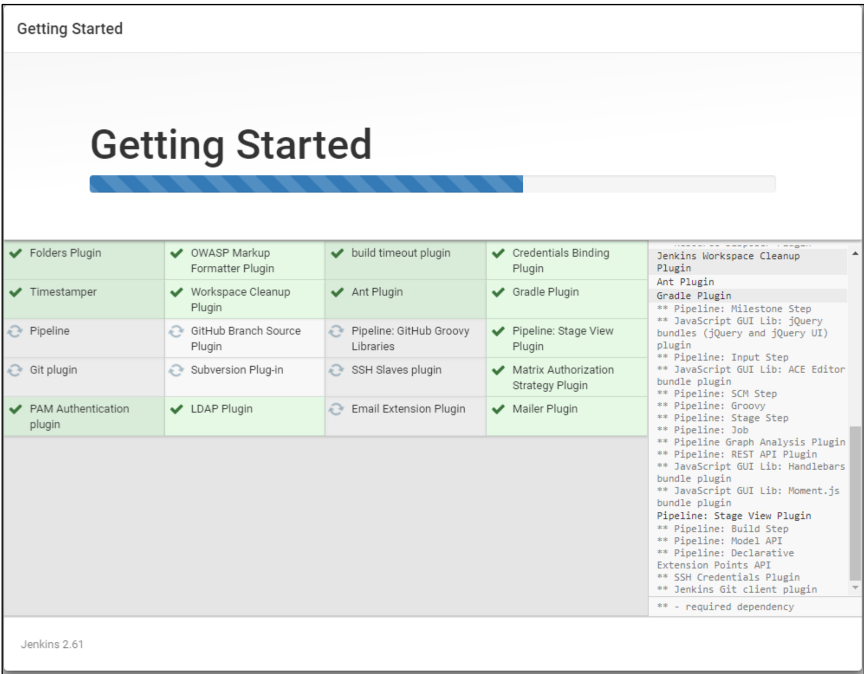
配置 Jenkins

现在该配置 Jenkins 了。我们从系统上的指定路径（即`/var/lib/Jenkins/secrets/initialAdminPassword`）获取密码，将其粘贴到安装向导的输入框中，然后单击 **Continue** 按钮。之后将会看到如下图所示的画面。



在下一个界面中，将看到可供安装的插件列表。我们将选择 **Install suggested Plugins**（**安装建议的插件**）选项。注意，也可以在初始配置之后安装额外的插件。所以，不用担心！

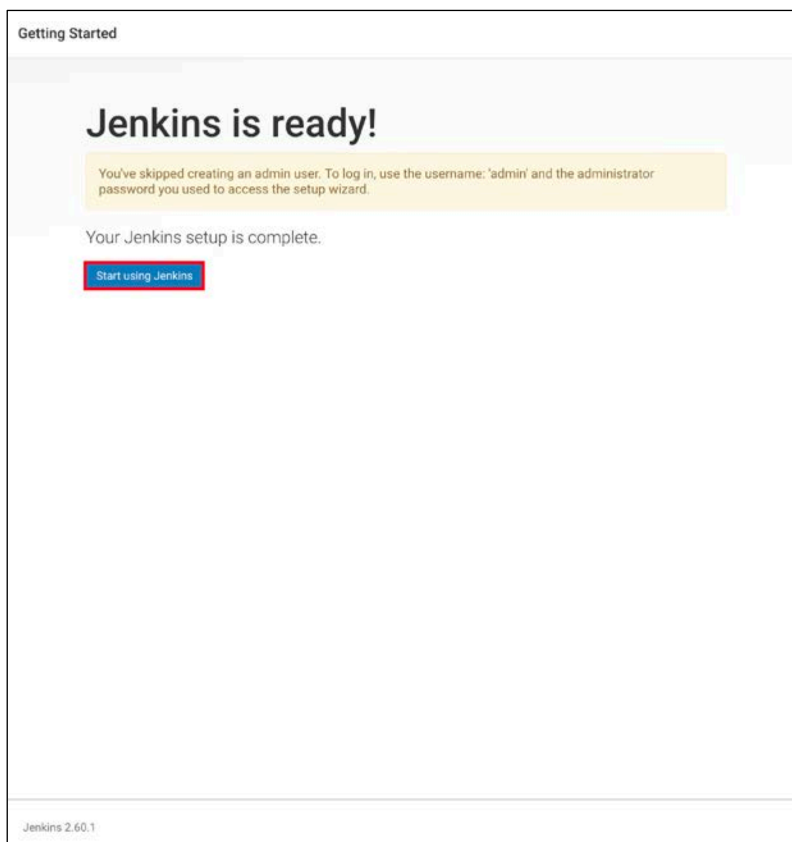
之后，你将看到下图所示的界面，显示当前的插件安装进度。



插件的安装可能需要花点时间。这些插件都是 Jenkins 建议安装的，因为在项目的 job 中可能需要依赖它们。

插件安装完成后，它会要求你创建一个用来访问 Jenkins 控制台的管理员用户。注意为了设置 Jenkins，我们使用了临时凭证，如下图所示。

输入用户信息后，单击 **Save and Finish**（**保存和完成**）按钮以完成设置，如下图所示。

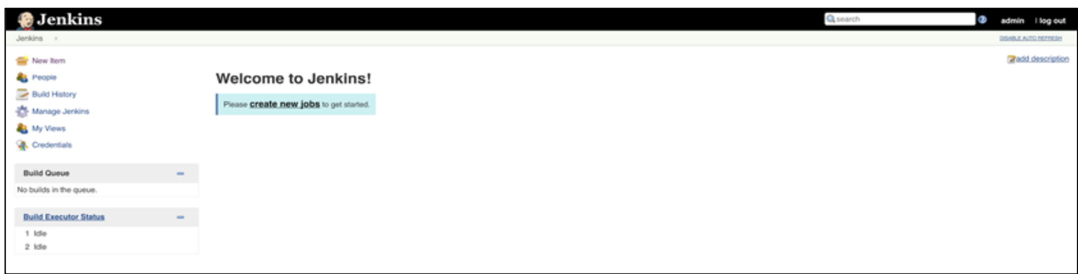


Jenkins 安装完成。

Jenkins 自动化配置

在本节中，我们来了解一下 Jenkins 不同部分的配置，并创建我们的第一个 job 及构建我们的应用程序。

在理想情况下，成功登录 Jenkins 后，主页应该看起来如下图所示。



Jenkins 安全配置

强烈建议设置 Jenkins 的安全性，以确保你的控制台安全，因为我们会将应用程序暴露给 Jenkins。

在 Jenkins 主页上，单击 **Manage Jenkins**（**管理 Jenkins**）以导航到 Jenkins 的设置部分，然后单击右侧面板中的 **Configure Global Security**（**配置全局安全性**）选项打开安全面板。

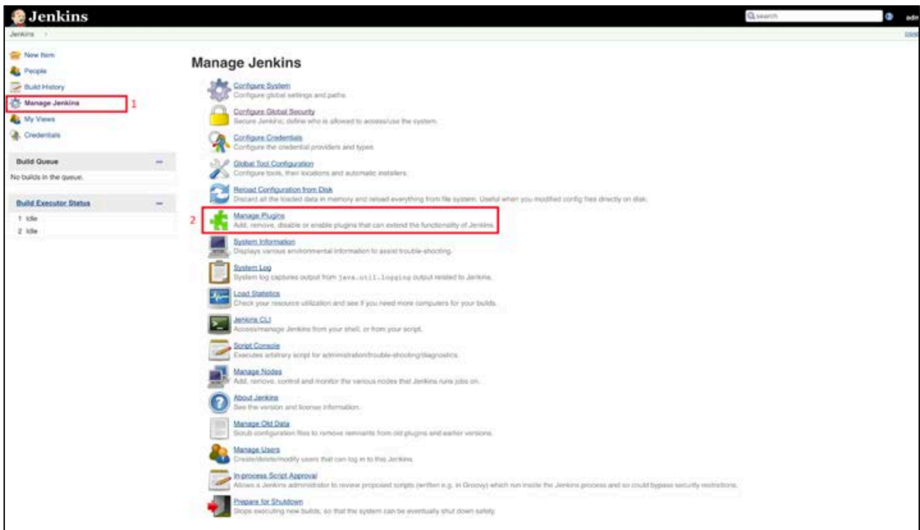
在 **Configure Global Security**（**配置全局安全性**）部分，可以管理用户授权，如下图所示。



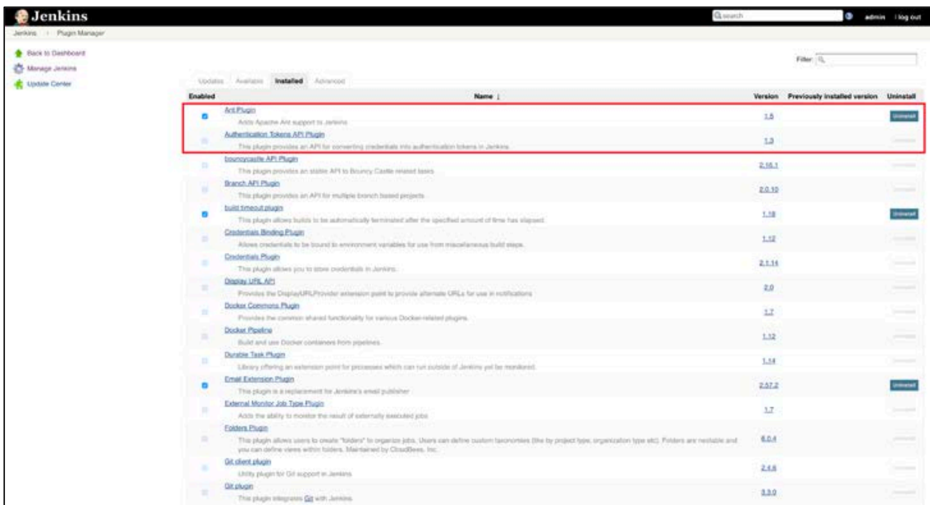
在这里，你可以根据用户的角色定义一个访问列表。一般来说，在大型组织中，用户访问权根据用户的使用情况提供给不同的人，这样 Jenkins 的安全性就可以保证。通常，可以使用基于 UNIX 的用户/组数据库或 Jenkins 自己的用户数据库来设置。

插件管理

插件管理是非常重要的，因为这些插件使我们能够将 Jenkins 与不同的环境（可能是云平台）或本地资源集成，也使我们能够管理自己资源上的数据，如应用程序服务器、数据库服务器等。



在 **Manage Jenkins**（**管理 Jenkins**）面板上，选择 **Manage Plugins**（**管理插件**）选项，打开 **Manage Plugins**（**插件管理**）面板，如下图所示。



在该面板中，你可以安装、卸载和升级系统中的指定插件。在该面板中也可以升级Jenkins。

版本控制系统

Jenkins 主要用于构建特定的应用程序代码，或者在任何基础架构平台上部署代码（即用于持续部署）。

现在，所有组织都会将应用程序代码存储在版本控制系统（如管理员具有中央控制权的 Git）中，并且可以根据用户角色提供所需的访问权限。另外，由于我们正在讨论持续集成，因此建议将应用程序代码存储在具有版本控制功能的集中位置，以保持代码的完整性。

因此，为了维护版本代码，请确保在 **Manage plugin panel**（**插件管理面板**）中安装了 Git 插件。

为了能够通过 Jenkins 克隆 Git 仓库，需要输入你的 Jenkins 系统的邮箱和用户名。为此，切换到你的工作目录，然后运行 Git config 命令，如下所示：

```
# Need to configure the Git email and user for the Jenkins job

# switch to the job directory
cd /var/lib/Jenkins/jobs/myjob/workspace

# setup name and email
sudo git config user.name "Jenkins"
sudo git config user.email "test@gmail.com"
```

当从代码库下载代码或者合并 Git 分支时需要设置。

设置 Jenkins job

现在我们可以开始设置第一个 Jenkins job 了。前面讨论过，每个 job 都用于指行特定的任务，可以是单个任务也可以是一个流水线。

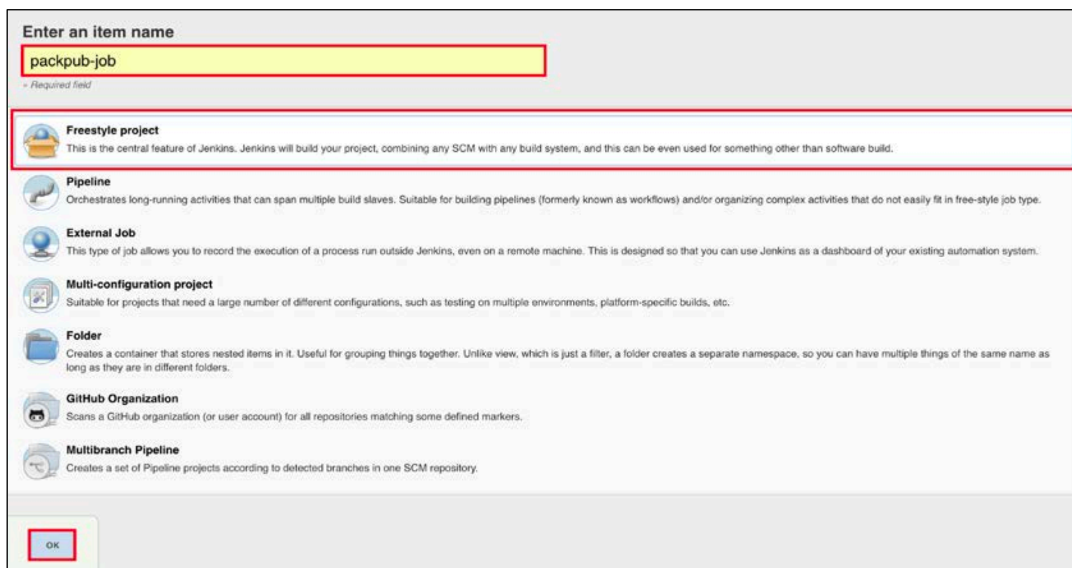
根据 Andrew Phillips 的说法，在理想情况下，流水线将软件交付过程分解成了几个阶段。每个阶段都旨在从不同角度来验证新功能的质量，并防止错误影响到用户。如果遇到

任何错误，则以报告形式反馈，确保达到我们所需的软件质量。

为了创建 job，在 Jenkins 的主页上，单击左边的 **New item**（**新建项目**）选项，或者单击右侧面板中的 **create new jobs**（**新建 job**）链接，如下图所示。



之后，将会弹出一个向导，询问你的项目/job 名称和想要创建的 job 类型，如下图所示。



每个 job 类型下面都有该类型的描述。你必须选择一种类型，因为每个类型都有不同的配置。

请注意，因为我们使用的是最新版的 Jenkins，因此可能某些类型在旧版本的 Jenkins 中还没有，所以请确保你已安装了最新版本的 Jenkins。

我们选择 **Freestyle**（**自由风格**）项目，指定一个唯一的 job 名称，然后单击 OK 按钮继续配置 job。之后，你将看到下图所示的页面。

在该页面中，你可以定义 job 的详细信息，如项目名称、描述、GitHub 项目等。

接下来单击 **Source Code Management**（源码管理）标签，你将会看到下图所示的画面。

在这里，你可以定义源代码的详细信息。如果你之前在配置部分没有设置 Jenkins 用户凭证，在这里依然可以设置，单击证书旁边的 Add 按钮。将会弹出一个窗口，如下图所示。



你在此定义的用户（即 admin）需要拥有代码库的访问权限。

为代码库中的用户设置身份验证的方式有好几种，如下图所示。

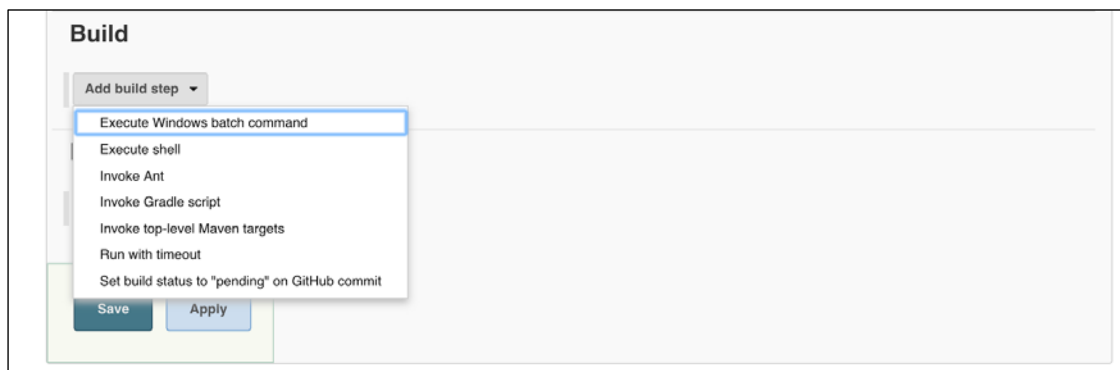
Jenkins 会立即根据你输入的存储库 URL 来测试凭证，这一点很重要。如果测试失败，则显示错误，如下图所示。

假设凭证与存储库 URL 匹配，则继续单击 **Build Triggers（生成触发器）**选项卡来滚动它。下图显示了可在 job 中用于持续部署的、强制执行的 **Build Triggers（生成触发器）**选项。

这个 **Build Triggers（构建触发器）**页面非常重要，因为它决定了构建运行的频率，以及触发构建的参数。例如，如果要在每次 Git 提交后都构建应用程序，则可以选择 **Build when a change is pushed to GitBucket（在变更推送到 GitBucket 时构建）**选项。

只要开发者提交任何变更到代码库中的某个分支（通常是 master）就会触发自动构建。这就像是你的代码库中的一个钩子，它持续跟踪代码库中的活动。另外，如果你想周期性地构建应用或者执行 job，可以在 **Poll SCM（轮训 SCM）**中指定条件，如 `H/15 * * * *` 来调度 job，这意味着该 job 将每隔 15 分钟运行一次。这与我们通常在基于 Linux 的系统中建立的 cron 作业类似。

接下来是 **Build environment（构建环境）**和 **Build（构建）**，这两部分是为工作空间相关的任务设置的。因为我们构建的是一个基于 Python 的应用程序，而且我们已经构建了应用程序，所以我们可以跳过这些部分的设置。但是如果你有一个使用 Java 或者.NET 编写的应用程序，那么你可以使用 ANT 和 Maven 构建工具，然后通过分支来构建。另外，如果你想建立一个基于 Python 的应用程序，那么可以试试 pyBuilder（`http://pybuilder.github.io/`）这样的工具。下图显示了构建选项。



完成后，你可以单击下一个选项卡，这是构建后操作。用于定义构建成功后要做的工作。我们讲过，Jenkins 也可以用作持续部署工具。因此，在构建后操作中，你可以指定将应用程序部署到什么平台，例如 AWS EC2 机器、Azure VM 或其他平台。

在构建后部分，在持续集成的前提下，还可以在构建成功后执行如 Git merge 或在 Git 上发布等操作。另外，还可设置你的利益相关者的电子邮箱，向他们发送电子邮件，以提供最新的构建结果。请参阅下图以了解详情。

Post-build Actions

Git Publisher

Push Only If Build Succeeds

Merge Results

If pre-build merging is configured, push the result back to the origin

Force Push

Add force option to git push

Tags [Add Tag]

Tags to push to remote repositories

Branches [Add Branch]

Branches to push to remote repositories

Notes [Add Note]

Notes to push to remote repositories

E-mail Notification

Recipients: manish@sethis.in

Whitespace-separated list of recipient addresses. May reference build parameters like $PARAM. E-mail will be sent when a build fails, becomes unstable or returns to stable.

- [x] Send e-mail for every unstable build
- [ ] Send separate e-mails to individuals who broke the build

Set build status on GitHub commit [deprecated]

[Advanced...]

[Add post-build action]

填写完所需的信息后，单击 **Save** 按钮保存配置。现在，我们已经为构建应用程序做好了准备——单击左侧面板中的 **Build Now**（**立即构建**）链接，如下图所示。





对于第一次构建执行，如果你尚未设置**轮询 SCM** 或**构建触发器**，则需要手动触发。

关于 Jenkins 的 job 创建方面，我们需要了解的就这么多。但是，在接下来的章节中，我们将使用 Jenkins 作为持续交付和持续集成工具，在这些章节中，我们将把在之前的章节中创建的 React 应用程序部署到不同平台（如 AWS、Azure 或 Docker）上。我们还将了解如何将 AWS 服务与 Jenkins 集成,通过一次提交就可以自动将应用程序交付到 GitHub 存储库。

理解持续交付

持续交付是一种软件工程实践，就是将生产就绪功能部署到生产中。

持续交付的主要目标是不管要交付的平台如何，都可以成功地完成应用程序部署，而这些平台可能是大型分布式系统或复杂的生产环境。

在跨国公司中，即使有许多开发人员同时操作不同的应用程序组件，也总是要确保应用程序代码处于稳定和可部署状态。在持续交付中，我们也需要确保单元测试和集成测试能够成功执行，这样才可以称为生产准备就绪。

持续交付的诉求

假设我们想要更频繁地部署软件，那么就不应该期望我们的系统具有较高的稳定性和可靠性，这种说法不完全正确。持续交付为那些愿意牺牲部分稳定性的组织提供了难以置信的竞争优势，同时也能够在竞争激烈的市场中生产出可靠的软件。

持续交付给我们带来了如下好处。

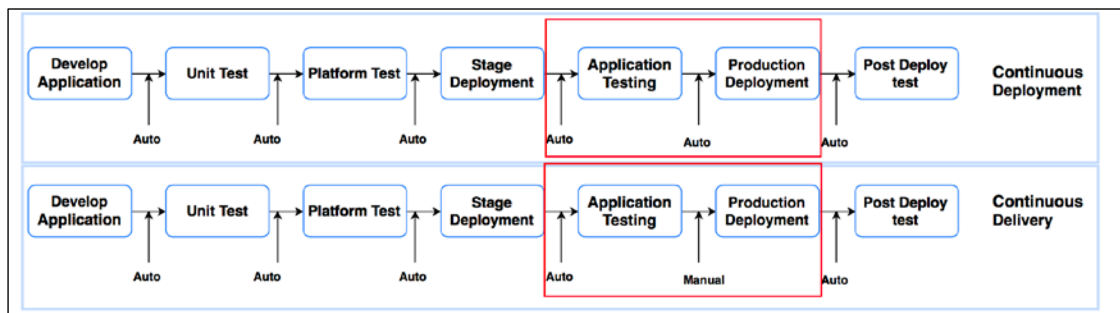
- **无风险发布**：任何版本的应用程序的主要诉求都是具有最小或零的停机时间。毕竟，软件是商业化的，用户不应该因为频繁发布而受到影响。通过使用蓝绿部署（`https://martinfowler.com/bliki/BlueGreenDeployment.html`）等模式，我们可以在部署期间实现零宕机。
- **竞争激烈的市场**：在持续交付中，所有团队（如构建和部署团队、测试团队、开发人员等）一起工作，使得测试、集成等不同活动天天都在发生。这使得功能发

布的过程更快（一周或两周），并且软件将被频繁地发布到生产环境以供客户使用。

- **质量改进**：在持续交付中，开发人员不必担心测试过程，因为这是由流水线维护的，并将结果展示给 QA 团队。这使得 QA 团队和开发人员能够深入了解探索性测试、可用性测试以及性能和安全性测试，从而来改善客户体验。
- **更好的产品**：通过在构建、测试、部署和环境设置中使用持续交付，降低了制造和交付软件增量更改的成本，使产品变得更好。

持续交付与持续部署

持续交付和持续部署在构建阶段和测试阶段，以及软件发布周期方面都是相似的，但是在过程方面略有不同，你可以通过下图来了解。



在持续部署中，生产就绪的代码一旦通过了所有的测试就可以直接部署到生产环境中，这使得软件的发布变得更加频繁。但在持续交付的情况下，除非有相关管理者手动触发或批准，否则不会将生产就绪的代码部署到生产环境中。

本章小结

在本章中，我们讨论了如 Jenkins 这样的 CI 和 CD 工具，还研究了它的功能。在此阶段了解这些工具非常重要，因为大多数使用云平台的公司都将这些流程用于软件开发和部署中。现在我们已经了解了部署流程，为了解用于部署应用程序的平台做好了准备。

在下一章中，我们将讨论 Docker（基于容器技术）。相信你们之前都听说过 Docker，所以请继续关注 Docker。下一章见！

10 应用容器化

在上一章中讨论了持续集成和持续交付/部署，这一章我们将探讨基于容器的技术，比如使用 Docker 来部署我们的应用程序。本章将介绍 Docker 及其功能，并展示如何在 Docker 上部署我们的云原生应用程序。

本章包括以下主题：

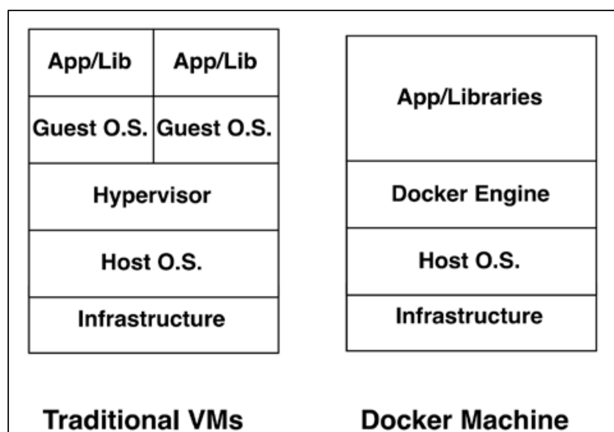
- 了解 Docker 以及它与虚拟化之间的不同
- 在不同的操作系统上安装 Docker 和 Docker Swarm
- 在 Docker 上部署云原生应用
- 使用 Docker Compose

Docker 介绍

Docker 是一个**容器管理系统**（**CMS**），借助于它你可以将应用程序与基础架构分离开来，从而更轻松地开发、发布和运行你的应用程序。Docker 对于管理 **Linux 容器**（**LXC**）非常有用。你可以使用它来创建镜像，对容器执行操作，在容器中执行命令。

简而言之，Docker 提供了一个平台，让你可以在称为容器的独立环境中打包和运行应用程序，然后在不同的软件发布环境（如阶段构建、预生产、生产等）中发布。

与任何传统的虚拟机相比，Docker 更轻量级，如下图所示。



关于 Docker 和虚拟化的一些事实

有很多组织仍然在传统 VM 上工作。有些组织已经将他们的应用程序迁移到了 Docker 上，或者已经准备迁移了。Docker 在以下几个方面比虚拟机更有潜力。

- Docker 比虚拟机的系统开销要低。
- Docker 环境中的应用程序比虚拟机具有更高的性能。
- 虚拟机软件技术名为 **Hypervisor**，它作为虚拟机环境和底层硬件之间的代理，提供必要的抽象层；在 Docker 中，我们有 Docker engine，相比于 Docker machine，它能提供更多的控制。
- 由于在 Docker 环境中共享了主机的操作系统，而虚拟机需要自己的操作系统来部署应用程序，这使得 Docker 与虚拟机相比更加轻量级并可以快速启动和销毁。Docker 与在主机操作系统上运行的其他进程相似。
- 对于云原生应用程序，我们需要在开发的每个阶段快速测试我们的微服务，而 Docker 平台就是一个很好的选择，可以用它来测试我们的应用程序，强烈推荐你这么做。

Docker Engine——Docker 的骨干

Docker Engine 是一个客户端/服务器应用程序，它具有以下组件。

- **Dockerd**：这是一个守护进程，它在主机操作系统的后台运行，以跟踪 Docker 容器属性，例如状态（启动、运行中、已停止）。

- **Rest API**：提供与守护进程之间交互的接口，以及操作容器的接口。
- **Docker 命令行**： 提供用来创建和管理 Docker 对象，如镜像、容器、网络和卷等的命令行界面

配置 Docker 环境

在本节中，我们介绍 Docker 在不同操作系统（如 Debian 和 Windows 等）上的安装过程。

在 Debian 上安装 Docker

设置 Docker 非常简单。市场上主要有两个版本的 Docker。

Docker 公司拥有容器化 Docker 产品，他们将 Docker **Commercially Supported**（**CS**）版更名为 Docker **Enterprises Edition**（**EE**）, 并将 Docker Engine 转换为 Docker **Community Edition**（**CE**）。

EE 和 CE 之间有几点不同，显然，商业上的支持是其中之一。但是，在 Docker 企业版中，Docker 公司已经围绕容器内容、平台插件等方面建立了一些认证。

在本书中，我们将使用 Docker Community Edition，所以我们首先更新 APT 存储库：

```
$ apt-get update -y
```

然后从 Docker 官方系统添加 GPG 密钥，如下所示：

```
$ sudo apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80 --recv-keys 58118E89F3A912897C070ADBF76221572C52609D
```

随后，将 Docker 存储库添加到 Ubuntu 的 APT 源列表中：

```
$ sudo apt-add-repository 'deb https://apt.dockerproject.org/repo ubuntu-xenial main'
```



有时，在 Ubuntu 14.04/16.04 中会找不到 `apt-add-repository` 工具。为了安装它，请使用以下命令安装 `software-properties-common` 软件包：

```
$ sudo apt-get install software-properties-common -y
```

接下来，更新 APT 软件包管理器，以下载最新的 Docker 列表：

```
$ apt-get update -y
```



如果要从 Docker 存储库（而不是默认的 14.04 存储库）下载并安装 Docker engine，请使用以下命令：

```
$ apt-cache policy docker-engine
```

你可以在终端中看到如下图所示的输出。

```
root@packtpub:~# apt-cache policy docker-engine
docker-engine:
  Installed: (none)
  Candidate: 17.05.0~ce-0~ubuntu-xenial
  Version table:
     17.05.0~ce-0~ubuntu-xenial 0
        500 https://apt.dockerproject.org/repo/ ubuntu-xenial/main amd64 Packages
     17.04.0~ce-0~ubuntu-xenial 0
        500 https://apt.dockerproject.org/repo/ ubuntu-xenial/main amd64 Packages
     17.03.1~ce-0~ubuntu-xenial 0
        500 https://apt.dockerproject.org/repo/ ubuntu-xenial/main amd64 Packages
     17.03.0~ce-0~ubuntu-xenial 0
        500 https://apt.dockerproject.org/repo/ ubuntu-xenial/main amd64 Packages
     1.13.1-0~ubuntu-xenial 0
        500 https://apt.dockerproject.org/repo/ ubuntu-xenial/main amd64 Packages
     1.13.0-0~ubuntu-xenial 0
        500 https://apt.dockerproject.org/repo/ ubuntu-xenial/main amd64 Packages
     1.12.6-0~ubuntu-xenial 0
        500 https://apt.dockerproject.org/repo/ ubuntu-xenial/main amd64 Packages
     1.12.5-0~ubuntu-xenial 0
        500 https://apt.dockerproject.org/repo/ ubuntu-xenial/main amd64 Packages
     1.12.4-0~ubuntu-xenial 0
        500 https://apt.dockerproject.org/repo/ ubuntu-xenial/main amd64 Packages
     1.12.3-0~xenial 0
        500 https://apt.dockerproject.org/repo/ ubuntu-xenial/main amd64 Packages
     1.12.2-0~xenial 0
        500 https://apt.dockerproject.org/repo/ ubuntu-xenial/main amd64 Packages
     1.12.1-0~xenial 0
        500 https://apt.dockerproject.org/repo/ ubuntu-xenial/main amd64 Packages
     1.12.0-0~xenial 0
        500 https://apt.dockerproject.org/repo/ ubuntu-xenial/main amd64 Packages
     1.11.2-0~xenial 0
        500 https://apt.dockerproject.org/repo/ ubuntu-xenial/main amd64 Packages
     1.11.1-0~xenial 0
        500 https://apt.dockerproject.org/repo/ ubuntu-xenial/main amd64 Packages
     1.11.0-0~xenial 0
        500 https://apt.dockerproject.org/repo/ ubuntu-xenial/main amd64 Packages
```

现在，我们已经准备好安装我们的 Docker Engine 了，在命令行中输入下面的命令：

```
$ sudo apt-get install -y docker-engine -y
```

由于 Docker 依赖于几个系统库，因此可能会遇到下图所示的错误。

```
root@packtpub:~# sudo apt-get install -y docker-engine
Reading package lists... Done
Building dependency tree
Reading state information... Done
Some packages could not be installed. This may mean that you have
requested an impossible situation or if you are using the unstable
distribution that some required packages have not yet been created
or been moved out of Incoming.
The following information may help to resolve the situation:

The following packages have unmet dependencies:
 docker-engine : Depends: init-system-helpers (>= 1.18~) but 1.14 is to be installed
                 Depends: lsb-base (>= 4.1+Debian11ubuntu7) but 4.1+Debian11ubuntu6 is to be installed
                 Depends: libdevmapper1.02.1 (>= 2:1.02.97) but 2:1.02.77-6ubuntu2 is to be installed
                 Depends: libltdl7 (>= 2.4.6) but it is not going to be installed
                 Depends: libsystemd0 but it is not installable
E: Unable to correct problems, you have held broken packages.
```

如果你遇到这种错误，那么请确保你已经安装了这些库。

Docker Engine 安装成功后，可以通过执行以下命令来验证：

```
$ docker –v
Docker version 17.05.0-ce, build 89658be
```

如果你看到如前所示的终端显示，那么可以继续进行下一步。

要获得 Docker 的帮助，可以执行以下命令：

```
$ docker help
```



如果你真的想使用 Docker 企业版，则可以继续执行 Docker 官方网站上的安装步骤（`https://docs.docker.com/engine/installation/linux/ubuntu/`）。

在 Windows 上安装

一般情况下，Windows 不适合 Docker，这就是为什么你没有在 Windows 系统上看到容器技术的原因。对此，我们有几个解决方法，其中之一是使用 Chocolatey。

在使用 Chocolatey 的 Windows 系统上安装 Docker，可按照下列步骤操作。

1. 从官方网站（`https://chocolatey.org/install`）安装 Chocolatey。

在该链接中显示了几种安装 Chocolatey 的方法。

2. 安装了 Chocolatey 后，你只需在 cmd 或 PowerShell 中执行以下命令：

```
$ choco install docker
```

这将在 Windows 7 和 8 操作系统上安装 Docker。

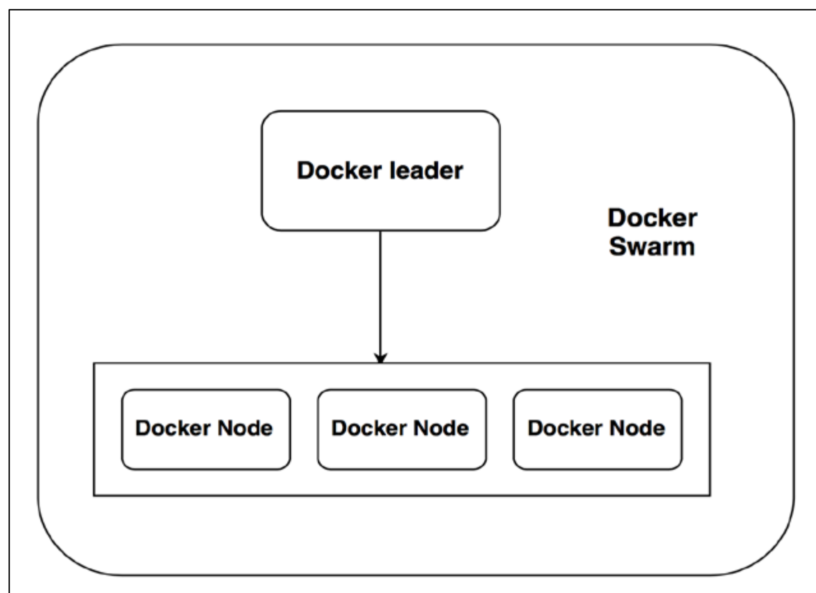
同样，如果你想要使用 Docker 企业版，则可以按照以下链接中显示的步骤操作：

https://docs.docker.com/docker-ee-for-windows/install/#install-docker-ee.Setting-up

Docker Swarm

Docker Swarm 是 Docker machine 池的一个流行术语。Docker Swarm 对托管网站非常有用，因为使用它可以快速缩放你的基础架构。

在 Docker Swarm 中，我们可以将多台 Docker 机器合并为一个单元，共享它们的资源，如 CPU、内存等，其中一台机器我们称之为 leader，其余节点作为 worker。



配置 Docker 环境

在本节中，我们将通过从 Docker 机器中选择 leader 并将其余机器与 leader 连接来设置 Docker Swarm。

假设

以下是对 Docker 环境做的一些假设：

- 出于演示的目的我们将两台机器（可能是来自云平台的虚拟机或实例）命名为 master 和 node1。另外，按照 Docker 安装部分中描述的过程在两台机器上安装了 Docker。
- master 和 node1 主机上的 2377 端口必须打开用于通信。
- 确保打开应用程序访问端口；在我们的例子中，为 nginx 的 80 端口。
- master Docker 机器可以基于任何类型的操作系统，比如 Ubuntu、Windows 等。

下面，我们来设置 Docker Swarm。

初始化 Docker manager

我们需要决定哪个节点作为 leader。我们选择 master 节点作为 Docker manager。因此，登录到该主机并执行以下命令初始化该主机使其成为 Docker Swarm 的 leader：

```
$ docker swarm init --advertise-addr master_ip_address
```

该命令将提供的主机设置为 master（leader），并生成一个供节点连接时使用的令牌。输出如下图所示。

```
root@ip-10-0-0-217:~# docker swarm init --advertise-addr 54.158.3.255
Swarm initialized: current node (zc0r8kzmjf7wa18h0m2i33u4b) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join \
    --token SWMTKN-1-1le69e43paf0vxyvjdslxaluk1a1mvi5lb6ftvxdoldul6k3dl-1dr9qdmbmni5hnn9y3oh1nfxp \
    54.158.3.255:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.
```

注意下面两点：

- 不要与任何人分享你的令牌和 IP 地址。
- 在故障转移的情况下可以有多个 master。

将 node1 添加到 master

选择了 leader 后，需要向集群添加一个新的节点以完成设置。登录到 node1，执行以上命令输出中指定的命令：

```
$ docker swarm join    --token SWMTKN-1-
1le69e43paf0vxyvjdslxaluk1a1mvi5lb6ftvxdoldul6k3dl-
1dr9qdmbmni5hnn9y3oh1nfxp   master-ip-address:2377
```

输出如下图所示。

```
root@ip-10-0-0-64:~# docker swarm join \
>     --token SWMTKN-1-1le69e43paf0vxyvjdslxaluk1a1mvi5lb6ftvxdoldul6k3dl-1dr9qdmbmni5hnn9y3oh1nfxp \
>     54.158.3.255:2377
This node joined a swarm as a worker.
root@ip-10-0-0-64:~#
```

这意味着我们的设置是成功的。我们来看看它是否被添加到了主 Docker machine 中。

执行下面的命令来验证：

```
$ docker node ls
```

输出如下：

```
root@ip-10-0-0-217:~# docker node ls
ID                            HOSTNAME          STATUS          AVAILABILITY          MANAGER STATUS
cycsh0kdfnyucfo1pr7ari8qw     ip-10-0-0-64      Ready           Active
zc0r8kzmjf7wa18h0m2i33u4b *   ip-10-0-0-217     Ready           Active                Leader
```

测试 Docker Swarm

现在我们已经部署了 Docker Swarm，可以在它上面运行一些服务了，比如 nginx 服务。在主 Docker machine 上执行以下命令，在 80 端口上启动 nginx 服务：

```
$ docker service create  --detach=false -p 80:80 --name webserver
nginx
```

该命令的输出应该如下图所示。

```
root@ip-10-0-0-217:~# docker service create  --detach=false -p 80:80 --name webserver nginx
vawxbicnq616gblt5cfo875mh
overall progress: 1 out of 1 tasks
1/1: running   [==================================================>]
verify: Waiting 1 seconds to verify that tasks are stable...
root@ip-10-0-0-217:~#
```

使用下面的 Docker 命令来查看服务是否运行了：

```
$ docker service ps webserver
```

该命令的输出应该如下图所示。

```
root@ip-10-0-0-217:~# docker service ps webserver
ID                  NAME                IMAGE               NODE                DESIRED STATE       CURRENT STATE              ERROR
   PORTS
u7oj15hcifpm        webserver.1         nginx:latest        ip-10-0-0-217       Running             Running about a minute ago
```

其他一些要验证的命令如下。

要验证哪些服务正在运行以及在哪个端口上运行，请使用以下命令：

$ docker service ls

如果你看到的输出如下图所示，那么很好，我们可以继续下面的操作。

```
root@ip-10-0-0-217:~# docker service ls
ID                  NAME                MODE                REPLICAS            IMAGE               PORTS
vawxbicnq616        webserver           replicated          1/1                 nginx:latest        *:80->80/tcp
```

要扩展服务的 Docker 实例，请使用以下命令：

```
$ docker service scale webserver=3
```

输出如下图所示。

```
root@ip-10-0-0-217:~# docker service scale webserver=3
webserver scaled to 3
root@ip-10-0-0-217:~# docker service ps webserver
ID                  NAME                IMAGE               NODE                DESIRED STATE       CURRENT STATE            ERROR
ORTS
u7oj15hcifpm        webserver.1         nginx:latest        ip-10-0-0-217       Running             Running 6 minutes ago
loqmks4894gx        webserver.2         nginx:latest        ip-10-0-0-64        Running             Running 34 seconds ago
5qw9xz92agl4        webserver.3         nginx:latest        ip-10-0-0-217       Running             Running 10 seconds ago
```

我们通过访问默认页面来检查 nginx 是否启动。在浏览器中访问 `http://master-ip-address80/`。如果看到如下图所示的输出，则说明你的服务已成功部署。

Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

真棒！下面，我们将在 Docker machine 上部署我们的云原生应用程序。

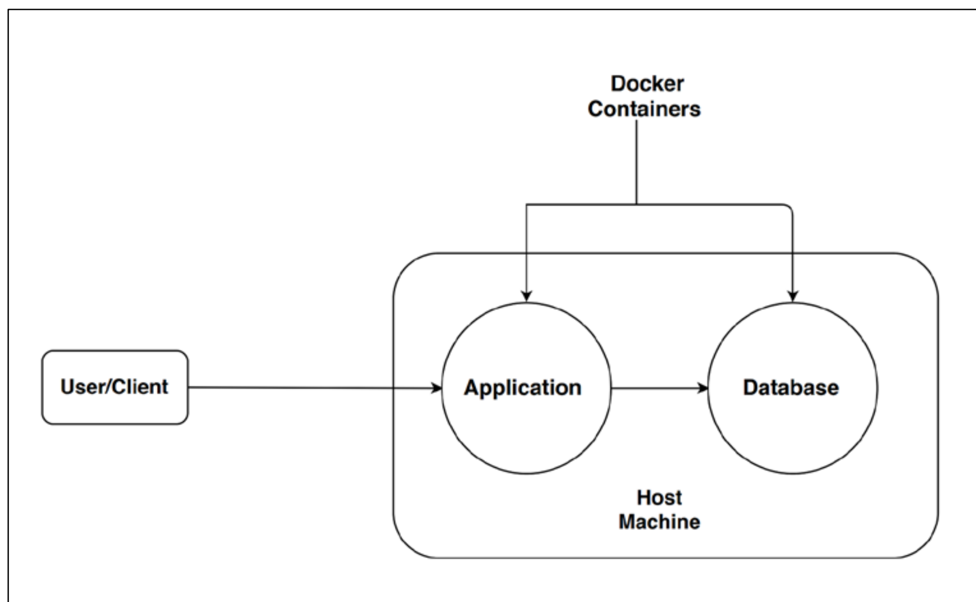
在 Docker 中部署应用

在本节中，我们将部署我们在前面章节中开发的云原生应用程序。但是，在开始创建应用程序基础架构之前，应该知道一些 Docker 的概念。

- **Docker 镜像**：基本上是库和在其上部署的应用程序的组合。这些镜像可以从 Docker Hub 公共存储库下载，也可以创建自定义镜像。
- **Dockerfile**：这是一个配置文件，用于构建运行 Docker machine 的镜像。
- **Docker Hub**：这是一个集中的存储库，你可以在这里保存镜像，其可以在整个团队中共享。

我们将在应用程序部署期间使用所有这些概念。此外，我们将继续使用 Docker Swarm 设置来部署应用程序，因为我们不想耗尽资源。

我们将遵循这个架构来部署应用程序，在这里我们将应用程序和 MongoDB（基本上是应用程序数据）部署在不同的 Docker 实例中，因为建议始终保持应用程序和数据的独立，如下图所示。



构建和运行 MongoDB Docker 服务

在本节中，我们将创建用于构建 MongoDB 的 Dockerfile，其中包含所有信息，例如基础镜像、要暴露的端口以及如何安装 MongoDB 服务等。

现在，我们登录到 Docker master（leader）账户，并使用以下内容创建名为 **Dockerfile** 的 Dockerfile：

```
# MongoDB Dockerfile
# Pull base image.
FROM Ubuntu
MAINTAINER Manish Sethi<manish@sethis.in>
# Install MongoDB.
RUN \
apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10 && \
echo 'deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart
dist 10gen' /etc/apt/sources.list.d/mongodb.list && \
apt-get update && \
apt-get install -y mongodb-org && \
rm -rf /var/lib/apt/lists/*
# Define mountable directories.
VOLUME ["/data/db"]
# Define working directory.
WORKDIR /data
# Define default command.
CMD ["mongod"]
# Expose ports.
EXPOSE 27017
EXPOSE 28017
```

保存。在继续执行后面的步骤之前，我们先来了解下该文件中的不同部分。

```
# Pull base image.
FROM ubuntu
```

这两行代码告诉你从 Docker Hub 中获取 Ubuntu 公共镜像，并将其作为运行以下命令的基础镜像：

```
# Install MongoDB
RUN \
apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10 && \
echo 'deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart
dist 10gen' > /etc/apt/sources.list.d/mongodb.list && \
apt-get update && \
apt-get install -y mongodb-org && \
rm -rf /var/lib/apt/lists/*
```

这段代码与我们为 MongoDB 手动执行的命令类似，但是在 Docker 中会自动处理这些命令。

接下来是 volume（存储卷）部分，这是可选的。创建可挂载的目录可以保证数据在外部卷中的安全。

```
# Define mountable directories.
VOLUME ["/data/db"]
```

下一部分是用户/客户端与 MongoDB 服务通信的端口：

```
EXPOSE 27017
EXPOSE 28017
```

保存文件，执行下面的命令创建镜像：

```
$ docker build --tag mongodb:ms-packtpub-mongodb
```



构建镜像大概需要 4～5 分钟时间，这取决于你的网络带宽和系统的性能。

下图显示了 Docker build 命令的输出。

```
root@ip-10-0-0-217:~/workspace/mongodb# docker build --tag mongodb:ms-packtpub-mongodb .
Sending build context to Docker daemon   2.56kB
Step 1/7 : FROM ubuntu
 ---> ebcd9d4fca80
Step 2/7 : RUN apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10 &&   echo 'deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart dist 10gen' > /etc/apt/sources.list.d/mongodb.list &&   apt-get update &&   apt-get install -y mongodb-org &&   rm -rf /var/lib/apt/lists/*
 ---> Using cache
 ---> 5a13063cfa01
Step 3/7 : VOLUME /data/db
 ---> Using cache
 ---> 23bb8d5ca556
Step 4/7 : WORKDIR /data
 ---> Using cache
 ---> f62ca2c0f725
Step 5/7 : CMD mongod
 ---> Using cache
 ---> 85e9910bccbd
Step 6/7 : EXPOSE 27017
 ---> Using cache
 ---> 09f21e252f59
Step 7/7 : EXPOSE 28017
 ---> Using cache
 ---> f24a8a6e986b
Successfully built f24a8a6e986b
Successfully tagged mongodb:ms-packtpub-mongodb
```

该图显示构建成功。查看镜像列表，看看是否存在我们标记的名称（**ms-packtpub-mongodb**）的镜像。

使用以下命令列出所有镜像：

```
$ docker images
```

下图显示的是所有可用的镜像。

```
root@ip-10-0-0-217:~/workspace/mongodb# docker images
REPOSITORY          TAG                    IMAGE ID            CREATED              SIZE
mongodb             ms-packtpub-mongodb    f24a8a6e986b        About a minute ago   411MB
ubuntu              latest                 ebcd9d4fca80        12 days ago          118MB
httpd               <none>                 e0645af13ada        2 weeks ago          177MB
nginx               <none>                 3448f27c273f        2 weeks ago          109MB
```

真棒！我们的镜像已经存在其中了。可以使用以下命令在 master Docker 机器上运行 mongodb 服务了：

```
$ docker run -d -p 27017:27017 -p 28017:28017 --name mongodb mongodb:ms-packtpub-mongodb mongod --rest --httpinterface
```

在输出中，我们将看到如下图所示的随机的 Docker ID。

```
root@ip-10-0-0-217:~/workspace/mongodb# docker run -d -p 27017:27017 -p 28017:28017 --name mongodb mongodb:ms-packtpub-mongodb mongod --rest --httpinterface
0e849fb79a486b22f882460cf4032f0182e8b503b29b653d0eeb3664fc364c5b
```

执行 docker ps 命令查看 Docker 容器的状态。应该会看到如下图所示的输出。

```
root@ip-10-0-0-217:~/workspace/mongodb# docker ps
CONTAINER ID        IMAGE                        COMMAND                  CREATED             STATUS              PORTS
                                        NAMES
0e849fb79a48        mongodb:ms-packtpub-mongodb   "mongod --rest --h..."   6 minutes ago       Up 6 minutes        0.0.0.0:27017->27017/
tcp, 0.0.0.0:28017->28017/tcp   mongodb
bb240462626a        nginx:latest                 "nginx -g 'daemon ..."   3 hours ago         Up 3 hours          80/tcp
                                        webserver.3.5qw9xz92agl4fzthw1j7lh7xr
1c6ed0dd5bd6        nginx:latest                 "nginx -g 'daemon ..."   3 hours ago         Up 3 hours          80/tcp
                                        webserver.1.u7oj15hcifpmu9frvfuxdx968
```

很少开发人员及系统管理员知道我们使用端口 28017 公开了 mongoDB 服务的 HTTP 接口。

因此，如果你在浏览器中访问 http://your-master-ip-address:28017/，将会看到如下图所示的界面。

mongod 0e849fb79a48

List all commands | Replica set status

Commands: buildInfo cursorInfo features hostInfo isMaster listDatabases replSetGetStatus serverStatus top

```
db version v2.6.12
git hash: d73c92b1c85703828b55c2916a5dd4ad46535f6a
sys info: Linux build5.ny.cbi.10gen.cc 2.6.32-431.3.1.el6.x86_64 #1 SMP Fri Jan 3 21:39:27 UTC 2014 x86_64 BOOST_LIB_VERSION=1_49
uptime: 511 seconds
```

overview (only reported if can acquire read lock quickly)

```
time to get readlock: 0ms
# databases: 2
# Cursors: 0
replication:
master: 0
slave:  0
```

clients

Client	OpId	Locking	Waiting	SecsRunning	Op	Namespace	Query	client	msg	progress
signalProcessingThread	0		{ waitingForLock: false }		0			:27017		
DataFileSync	1		{ waitingForLock: false }		0			:27017		
initandlisten	11		{ waitingForLock: false }		2002	local.startup_log		0.0.0.0:0		
journal	3		{ waitingForLock: false }		0			:27017		
RangeDeleter	6		{ waitingForLock: false }		0			:27017		
TTLMonitor	28		{ waitingForLock: false }		2004	local.system.indexes	{ expireAfterSeconds: { $exists: true } }	0.0.0.0:0		
clientcursormon	8		{ waitingForLock: false }		0			:27017		
snapshotthread	9		{ waitingForLock: false }		0			:27017		
websvr	12		{ waitingForLock: false }		0	admin.system.users		:27017		

dbtop (occurences|percent of elapsed)

NS	total		Reads		Writes		Queries		GetMores		Inserts		Updates		Removes	
TOTAL	3	0.0%	3	0.0%	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%
	3	0.0%	3	0.0%	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%

write lock % time in write lock, by 4 sec periods
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
write locked now: false

Log

```
2017-05-28T08:03:13.798+0000 [initandlisten] MongoDB starting : pid=1 port=27017 dbpath=/data/db 64-bit host=0e849fb79a48
           08:03:13.798+0000 [initandlisten] db version v2.6.12
           08:03:13.798+0000 [initandlisten] git version: d73c92b1c85703828b55c2916a5dd4ad46535f6a
           08:03:13.798+0000 [initandlisten] build info: Linux build5.ny.cbi.10gen.cc 2.6.32-431.3.1.el6.x86_64 #1 SMP Fri Jan 3 21:39:27 UTC 2014 x86_64 BOOST_LIB_VERSION=1_49
           08:03:13.798+0000 [initandlisten] allocator: tcmalloc
           08:03:13.798+0000 [initandlisten] options: { net: { http: { RESTInterfaceEnabled: true, enabled: true } } }
           08:03:13.800+0000 [initandlisten] journal dir=/data/db/journal
           08:03:13.800+0000 [initandlisten] recover : no journal files present, no recovery needed
           08:03:13.952+0000 [initandlisten] allocating new ns file /data/db/local.ns, filling with zeroes...
           08:03:14.005+0000 [FileAllocator] allocating new datafile /data/db/local.0, filling with zeroes...
           08:03:14.005+0000 [FileAllocator] creating directory /data/db/_tmp
           08:03:14.008+0000 [FileAllocator] done allocating datafile /data/db/local.0, size: 64MB,  took 0.001 secs
           08:03:14.009+0000 [initandlisten] build index on: local.startup_log properties: { v: 1, key: { _id: 1 }, name: "_id_", ns: "local.startup_log" }
           08:03:14.009+0000 [initandlisten]      added index to empty collection
           08:03:14.010+0000 [initandlisten] waiting for connections on port 27017
           08:03:14.010+0000 [websvr] admin web console waiting for connections on port 28017
           08:03:32.140+0000 [websvr] Socket recv() errno:11 Resource temporarily unavailable 103.252.26.196:55187
           08:04:13.954+0000 [clientcursormon] mem (MB) res:46 virt:351
           08:04:13.954+0000 [clientcursormon]  mapped (incl journal view):160
           08:04:13.954+0000 [clientcursormon]  connections:0
           08:09:13.967+0000 [clientcursormon] mem (MB) res:49 virt:351
           08:09:13.967+0000 [clientcursormon]  mapped (incl journal view):160
           08:09:13.967+0000 [clientcursormon]  connections:0
```

真棒！我们的 MongoDB 服务已经启动并运行了！

在继续启动应用程序的容器之前，我们先来了解一下 Dcoker Hub 的用途。

Docker Hub 是用来干什么的

根据 Docker Hub 官方文档的描述，Docker Hub 是一个基于云的注册表服务，其允许你

链接到代码库，构建镜像并测试，存储手动推送的镜像以及链接到 Docker Cloud，以便可以将镜像部署到主机。

简而言之，在 Docker Hub 中集中存储着任何人都可以访问的镜像，只要你具有所需的权限就可以执行镜像操作以在主机上部署和运行应用程序。

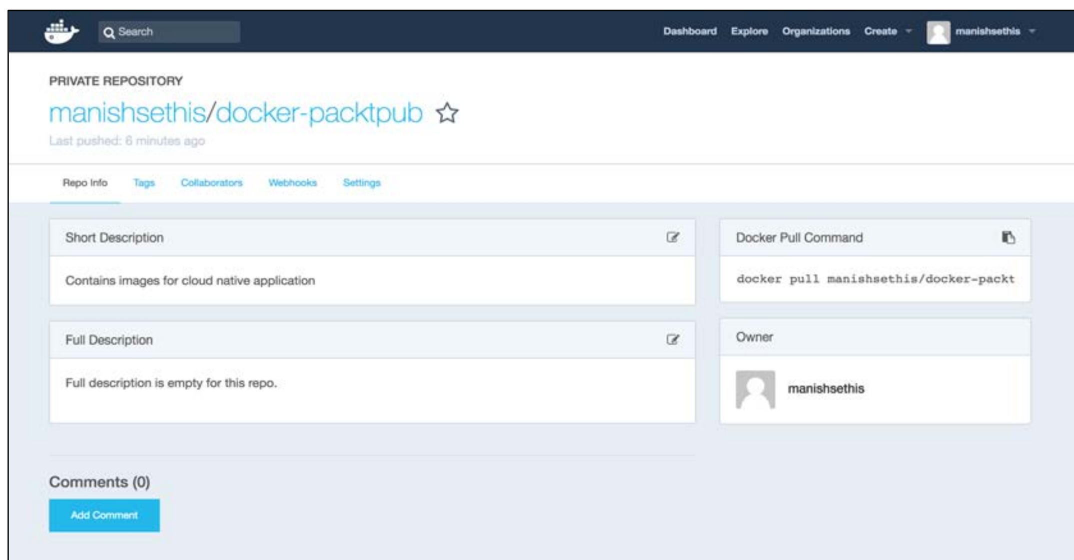
Docker Hub 有如下几个优点：

- Docker Hub 提供当代码库中有任何更改时自动构建镜像的功能。
- 提供 WebHook，在代码被成功推送到代码库中时自动触发应用程序部署。
- 提供创建私有空间的功能，该私有空间用来存储仅供组织或团队访问的镜像。
- Docker Hub 可以与版本控制系统集成，例如 GitHub、BitBucket 等，这对于持续集成和交付非常有用。

下面，我们来看看如何将自定义的 MongoDB 镜像推送到最新创建的私有存储库中。

首先，需要在 `https://hub.docker.com` 上创建一个账户并将其激活。登录后，你需要根据你的偏好创建一个私有/公共存储库，如下图所示。

单击 **Create** 按钮设置存储库，你将被重定向到下图所示的屏幕。





Docker Hub 为每个免费账户只提供一个私有存储库。

现在我们已经创建了存储库，我们回到 master Docker 机器并执行以下命令：

```
$ docker login
```

此命令会询问你的 Docker Hub 账户的凭证，如下图所示。

```
root@ip-10-0-0-217:~/workspace/mongodb# docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't have a Docker ID, head over to https://hub.docker.com to
 create one.
Username: manishsethis
Password:
Login Succeeded
```

登录成功后，可以使用以下命令为要推送到存储库的镜像添加标签：

```
$ docker tag mongodb:ms-packtpub-mongodb manishsethis/docker-packtpub
```



如果没有指定标签，那么它将默认采用 latest 的标签。

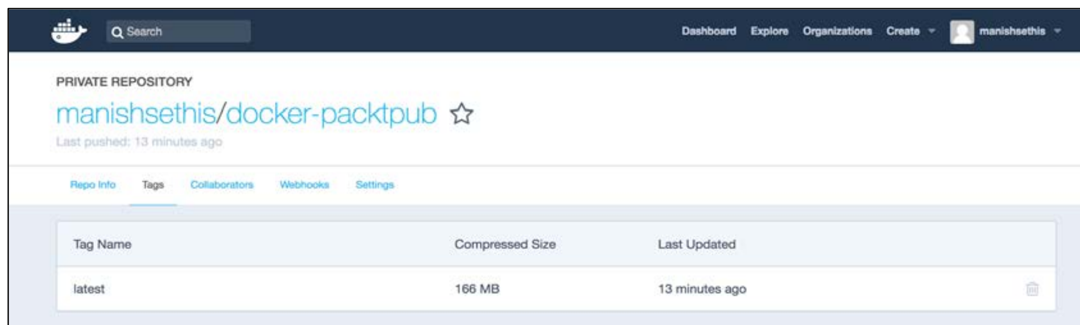
创建标签后就可以把标签推送到存储库。使用以下命令来执行此操作：

```
$ docker push manishsethis/docker-packtpub
```

下图显示了 Docker push 命令的输出。

```
root@ip-10-0-0-217:~/workspace/mongodb# docker push manishsethis/docker-packtpub
The push refers to a repository [docker.io/manishsethis/docker-packtpub]
a4c9b6de5784: Pushed
532078e28fc7: Pushed
33f1a94ed7fc: Mounted from library/ubuntu
b27287a6dbce: Mounted from library/ubuntu
47c2386f248c: Mounted from library/ubuntu
2be95f0d8a0c: Mounted from library/ubuntu
2df9b8def18a: Mounted from library/ubuntu
latest: digest: sha256:f99835f0d7c7178bc69afc5f5aebc067dbfee08b4409a988365a3f20ffee0589 size: 1776
```

完成推送后，将在 **Tags** 选项卡中看到 Docker Hub 中的镜像，如下图所示。



这意味着你的镜像已经被成功推送了。

要拉取镜像只需要使用下面的命令：

```
$ docker pull manishsethis/docker-packtpub
```

哇！这真是太简单了，只要你有凭证，就可以从任何地方访问。

还有其他 Docker 注册表提供程序，如 AWS（EC2 容器注册表）、Azure（Azure 容器注册表）等。

目前来说，这就是 Docker Hub 所带给我们的。本章将继续使用 Docker Hub 来推送镜像。

现在，我们已经准备好将云原生应用程序部署到另一个容器，但在此之前，需要使用 Dockerfile 为其构建镜像。因此，需要创建一个名为 app 的目录，创建一个 Dockerfile 并填入以下内容：

```
FROM ubuntu:14.04
MAINTAINER Manish Sethi<manish@sethis.in>
# no tty
ENV DEBIAN_FRONTEND noninteractive
# get up to date
RUN apt-get -qq update --fix-missing
# Bootstrap the image so that it includes all of our dependencies
RUN apt-get -qq install python3  python-dev python-virtualenv
python3-pip --assume-yes
RUN sudo apt-get install build-essential autoconf libtool libssl-
dev libffi-dev --assume-yes
# Setup locale
RUN export LC_ALL=en_US.UTF-8
RUN export LANG=en_US.UTF-8
RUN export LANGUAGE=en_US.UTF-8
# copy the contents of the cloud-native-app(i.e. complete
application) folder into the container at build time
COPY cloud-native-app/ /app/
# Create Virtual environment
RUN mkdir -p /venv/
RUN virtualenv /venv/ --python=python3
# Python dependencies inside the virtualenv
RUN /venv/bin/pip3 install -r /app/requirements.txt
# expose a port for the flask development server
EXPOSE 5000
# Running our flask application
CMD cd /app/ && /venv/bin/python app.py
```

尽管我们在前面已经解释了 Dockerfile 中的不同部分，但是还有一些部分需要解释一下。

```
COPY cloud-native-app/ /app/
```

Dockerfile 中的这行代码将应用程序的本地代码复制到 Docker 容器中。也可以使用 ADD 命令完成同样的功能。

CMD 是 command（命令）的缩写，用于在 Docker 容器中执行命令，在 Dockerfile 里这样定义它：

```
# Running our flask application
CMD cd /app/ && /venv/bin/python app.py
```

现在，保存文件，执行下列命令创建镜像：

```
$ docker build --tag cloud-native-app:latest .
```

该命令执行完成需要花一段时间，因为在这个过程中需要编译和安装很多库。最好是在每次代码更改后同时编译镜像。输出如下图所示。

```
root@ip-10-0-0-217:~/workspace/mongodb# docker build --tag cloud-native-app:latest .
Sending build context to Docker daemon   2.56kB
Step 1/7 : FROM ubuntu
 ---> ebcd9d4fca80
Step 2/7 : RUN apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10 &&   echo 'deb http://downloads-distro.mongodb.org/
repo/ubuntu-upstart dist 10gen' > /etc/apt/sources.list.d/mongodb.list &&   apt-get update &&   apt-get install -y mongodb-org &&   rm -
rf /var/lib/apt/lists/*
 ---> Using cache
 ---> 5a13063cfa01
Step 3/7 : VOLUME /data/db
 ---> Using cache
 ---> 23bb8d5ca556
Step 4/7 : WORKDIR /data
 ---> Using cache
 ---> f62ca2c0f725
Step 5/7 : CMD mongod
 ---> Using cache
 ---> 85e9910bccbd
Step 6/7 : EXPOSE 27017
 ---> Using cache
 ---> 09f21e252f59
Step 7/7 : EXPOSE 28017
 ---> Using cache
 ---> f24a8a6e986b
Successfully built f24a8a6e986b
Successfully tagged cloud-native-app:latest
```

请确保构建过程中的每一步都是成功的。

现在我们已经准备好了镜像，可以使用最新镜像启动容器了。

执行下列命令启动容器，记住要将 5000 端口暴露出来，这样才能从外部访问容器：

```
$ docker run -d -p 5000:5000  --name=myapp  cloud-native-app:latest
```

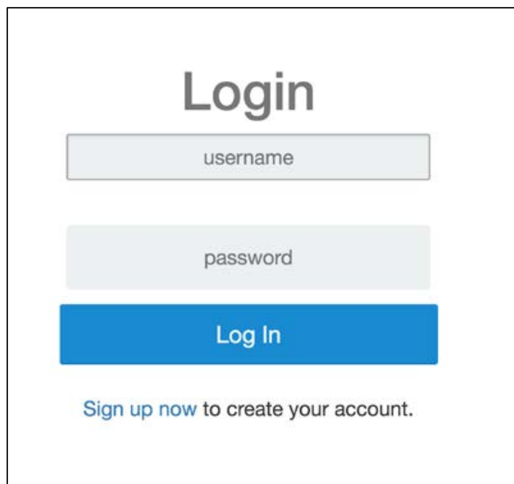
执行 `docker ps` 命令检查容器状态，输出如下图所示。

```
root@ip-10-0-0-217:~/workspace/mongodb# docker ps
CONTAINER ID        IMAGE                            COMMAND                  CREATED              STATUS              PORTS
                        NAMES
99aeb9ef94e8        177f23e53413                     "/bin/sh -c 'cd /a..."   35 minutes ago       Up 35 minutes       0.0.0.0:5000->5000/tc
p                       myapp
69a23999fefa        mongodb:ms-packtpub-mongodb      "mongod --rest --h..."   About an hour ago    Up About an hour    0.0.0.0:27017->27017/
tcp, 0.0.0.0:28017->28017/tcp   mongodb
```

如你所见，有两个容器正在运行，一个是 myapp 容器，一个是 mongodb 容器。

接下来，检查应用程序的 URL（`http://your-master-ip-address:5000/`）。

如果你看到如下图所示的界面，表示你的程序已经在 Docker 中成功运行。



现在我们可以创建新用户并登录，然后发布推文来测试应用程序。这里不再重复了，因为在创建应用程序时我们已经做了这些。

根据经验，应用程序和数据库（也就是 MongoDB）之间的通信可能会有一些问题，因为应用程序和数据库位于不同的容器中，它们可能位于独立的网络中。为了解决这个问题，可以创建一个网络，并将两个容器接入该网络。

例如，对于我们的容器（myapp 和 mongodb），可以按照以下步骤操作。

1. 使用下面的命令创建一个独立的网络：

```
$ docker network create -d bridge --subnet 172.25.0.0/16 mynetwork
```

2. 网络创建好之后，使用以下命令将这两个容器添加到这个网络中：

```
$ docker network connect mynetwork  myapp
$ docker network connect mynetwork  mongodb
```

3. 为了找到分配给这些容器的 IP，可以使用以下命令：

```
$ docker inspect --format '{{ .NetworkSettings.IPAddress }}'
$(docker ps -q)
```



创建网络是建立应用程序与数据库之间通信的一种方式。

好了，我们已经在 Docker 上部署了应用程序，并学习了它的概念。下面我们来学习 Docker Compose。让我们来看看它有什么不同之处吧！

Docker Compose

根据 Docker Compose 网站（`https://docs.docker.com/compose/overview/`）的官方说法，Compose 是定义和运行包含多个 Docker 容器的应用程序的工具。你可以使用 Compose 文件来配置你的应用程序的服务。

简而言之，它可以帮助我们更快速和简单地构建和运行应用程序。

在上一节中，我们在部署应用程序和构建镜像的过程中，首先创建了一个 Dockerfile，然后执行 `docker build` 命令来构建它。一旦构建完成，我们通常使用 `docker run` 命令启动容器，但是，在 Docker Compose 中，我们将定义一个带有详细配置信息的 `.yml` 文件，其中包括端口、要执行的命令等。

首先，Docker Compose 是一个独立于 Docker Engine 的实用程序。单击下面的链接，根据你的操作系统类型选择对应的版本安装：

`https://docs.docker.com/compose/install/.`

安装完成后，我们使用 Docker Compose 来运行容器。假设我们要使用 Docker Compose 运行云原生应用程序。我们已经为它生成了 Dockerfile，并且也指定应用程序位于相同的位置（路径）。

接下来，在 Dockerfile 所在的同一位置创建一个 `Docker-compose.yml` 文件，其中包括以下内容：

```
#Compose.yml
version: '2'
services:
web:
 build: .
```

```
      ports:
      - "5000:5000"
      volumes:
       - /app/
    flask:
  image: "cloud-native-app:latest"
```

在 docker-compose.yml 中添加好了配置后，保存，然后执行 docker-compose up 命令。在构建镜像过程中，我们可以看到如下图所示的输出。

```
root@ip-10-0-0-217:~/workspace/app# docker-compose up
WARNING: The Docker Engine you're using is running in swarm mode.

Compose does not use swarm mode to deploy services to multiple nodes in a swarm. All containers will be scheduled on the current node.

To deploy your application across the swarm, use `docker stack deploy`.

Starting app_flask_1 ...
Starting app_web_1 ...
Starting app_flask_1
Starting app_web_1 ... done
Attaching to app_flask_1, app_web_1
web_1    | * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
flask_1  | * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
web_1    | * Restarting with stat
flask_1  | * Restarting with stat
web_1    | * Debugger is active!
web_1    | * Debugger PIN: 190-114-053
flask_1  | * Debugger is active!
flask_1  | * Debugger PIN: 994-542-714
web_1    | 103.252.26.196 - - [28/May/2017 17:47:29] "GET / HTTP/1.1" 200 -
```

另外，如果你检查容器的状态，可以通过 Compose 启动多个容器（在我们的例子中是 app_web-1 和 app_flask_1），这就是为什么它对需要大规模基础设施的多容器应用程序很有用的原因，因为它创建了一个类似于 Docker Swarm 的 Docker 机器集群。下图显示了 Docker 机器的状态。

```
CONTAINER ID        IMAGE                       COMMAND                  CREATED             STATUS              PORTS
                          NAMES
7f8649ad3863        app_web                     "/bin/sh -c 'cd /a..."   3 minutes ago       Up About a minute   0.0.0.0:5000->5000/tc
p                         app_web_1
4d7d59ae4978        cloud-native-app:latest     "/bin/sh -c 'cd /a..."   6 minutes ago       Up About a minute   5000/tcp
                          app_flask_1
69a23999fefa        mongodb:ms-packtpub-mongodb "mongod --rest --h..."   3 hours ago         Up 13 minutes       0.0.0.0:27017->27017/
tcp, 0.0.0.0:28017->28017/tcp   mongodb
```

真棒！我们通过 Docker-compose 部署了我们的应用程序。现在，可以尝试访问应用程序的公用 URL（http://your-ip-address5000）以确认应用程序部署成功。

最后，确保将镜像推送到 Docker Hub 镜像存储仓库中。我们已经推送了 MongoDB 镜像。接下来使用以下命令推送我们的云原生应用镜像：

```
$ docker tag cloud-native-app:latest manishsethis/docker-packtpub:cloud-native-app
$ docker push manishsethis/docker-packtpub:cloud-native-app
```

应该可以看到如下图所示的输出。

```
root@ip-10-0-0-217:~/workspace/mongodb# docker tag cloud-native-app:latest manishsethis/docker-packtpub:cloud-native-app
(reverse-i-search)`': ^C
root@ip-10-0-0-217:~/workspace/mongodb# docker push manishsethis/docker-packtpub:cloud-native-app
The push refers to a repository [docker.io/manishsethis/docker-packtpub]
1e756c41967f: Pushed
03551bc2634a: Pushed
6feb34ef35c6: Pushed
21761af77f90: Pushed
50931687bd6d: Pushed
f160490bd2ed: Pushed
ec3db9fc1b33: Pushed
776d5289b76e: Mounted from library/ubuntu
0fb55a72eab2: Mounted from library/ubuntu
a30ab2bcda94: Mounted from library/ubuntu
99840408c5ea: Mounted from library/ubuntu
a8e78858b03b: Mounted from library/ubuntu
cloud-native-app: digest: sha256:99f80e75dd377db5a45b806a6ed8ef6a5f201f14d399b4888f112170757c4d2f size: 2833
```

本章小结

在本章中，我们学习了当前最有影响力的基于容器的技术 Docker。了解了 Docker 的一些概念并使用 Docker 部署了我们的应用程序。还探索了使用 Docker Compose 和 Dockerfile 来部署应用程序的方法。

在下一章中，我们将把应用程序部署到云平台，基于云平台来构建并部署应用程序的基础架构，这将会更有趣。你准备好了吗？我们下一章见！

11

部署到 AWS 云平台

在上一章中，我们介绍了 Docker 应用程序运行平台。可以使用它来隔离应用程序和来自客户的请求。在本章中，我们将介绍云平台，特别是主要用来处理 IaaS（基础架构即服务）和 PaaS（平台即服务）的 AWS（Amazon Web Services）。本章还会向你介绍构建基础设施和部署应用程序的过程。

本章包含如下主题：

- AWS 服务
- 使用 Terraform/CloudFormation 构建应用程序基础架构
- 使用 Jenkins 进行持续部署

AWS 入门

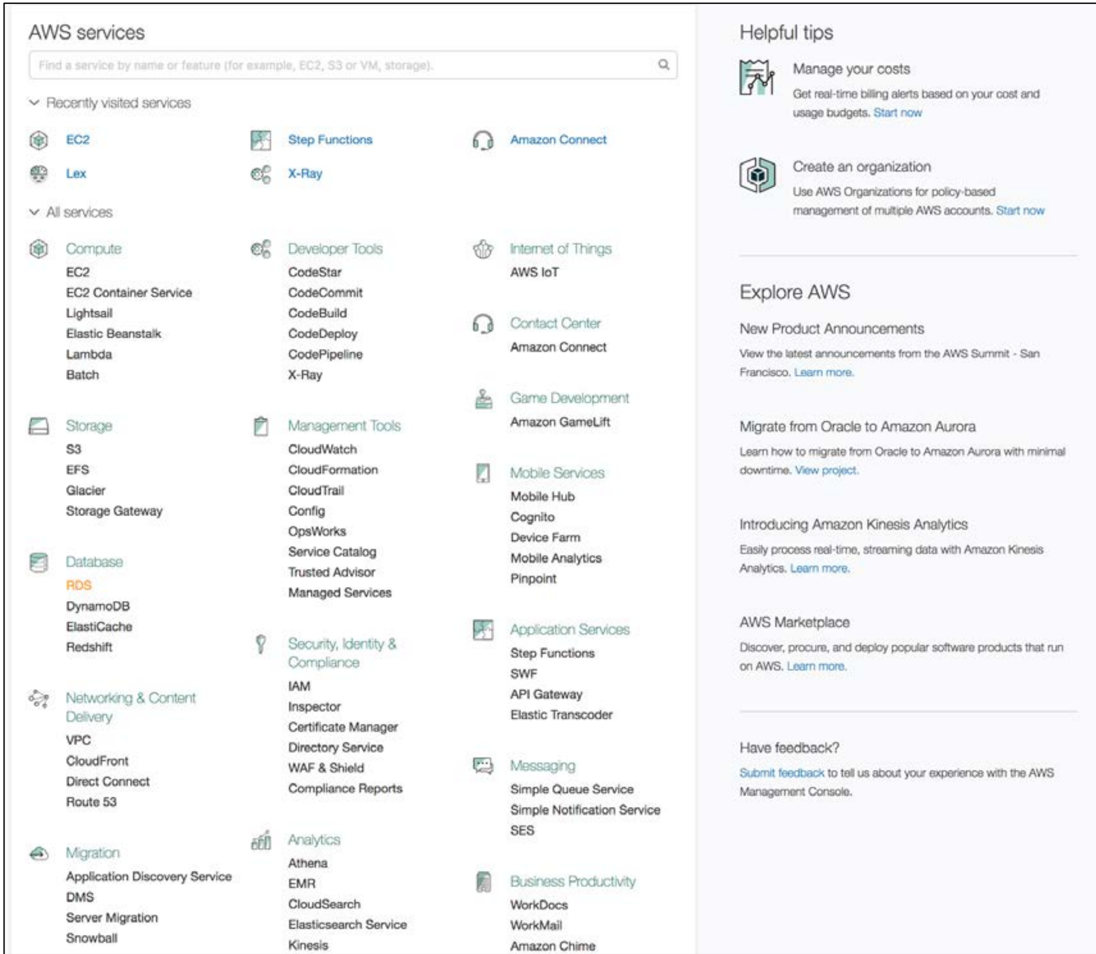
亚马逊网络服务（AWS）是一个安全的云平台。它在 IaaS 和 PaaS 中提供了各种产品，包括计算能力、数据库存储和内容交付，这些都有助于扩展应用程序，并使我们的业务在全球范围内增长。AWS 是一个公共云，按照云计算的概念，它使用按需付费的计划来按需交付资源。

在这里了解关于 AWS 服务的更多信息：`https://aws.amazon.com/`。

如前面一章中介绍的，你需要创建 AWS 账户才能创建服务。你可以通过下面的链接创建用户：

```
https://medium.com/appliedcode/setup-aws-account-1727ce89353e
```

登录后你会看到如下图所示的界面，其中显示了 AWS 的类别。有些服务正处于测试阶段。我们将使用一些与**计算**和**网络**相关的服务来构建应用程序的基础架构。



应用程序常用的 AWS 服务如下。

- **EC2 (Elastic compute cloud)**：这是 AWS 的计算产品，简而言之，其提供一个服务器。
- **ECS (Elastic Container Services)**：Amazon 上的 Docker 服务。它只能管理 EC2 机器上的 Docker。无须在本地创建 Docker 集群，你可以在几分钟内轻松地创建开销更少的 Docker。

- **EBS (Elastic bean stalk)**：这是一个 PaaS 产品，你只需要上传代码，并指定需要多少基础设施（基本上是应用程序服务器（EC2））。EBS 将创建机器，并部署代码。
- **S3 (Simple storage service)**：这是 AWS 提供的存储服务，通常用来存储应用程序的数据和静态内容，如托管静态网站。我们将用它来做持续部署。
- **Glacier**：这是另外一种存储服务，只用来做备份，因为它的成本比较低，因此与 S3 相比它的数据存储和检索能力比较低。
- **VPC (Virtual Private Network)**：这是一个网络服务，可以用来控制资源的访问权限。我们将使用这项服务来保护我们的应用程序服务和数据库，只向外界公开所需的资源。
- **CloudFront**：这是一个将你的 S3 中的内容分发到全球的内容分发服务，不管从哪里发送请求，它会确保资源都可以被检索到。
- **CloudFormation**：这为开发人员和系统管理员提供了一种创建和管理 AWS 上资源的简单方法，例如配置并通过代码来更新它们。我们将使用这项服务来创建我们的基础设施。
- **CloudWatch**：此服务用来跟踪资源的活动。它还会以日志的形式跟踪你在 AWS 账户上的任何活动，这对识别任何可疑活动和防止账户泄露非常有用。
- **IAM (Identity and Access Management)**：顾名思义，此服务就是用来管理 AWS 账户中的用户的，并根据用户的情况和要求为他们提供角色和权限。
- **Route 53**：这是一个高可用和可扩展的云 DSN 服务。我们可以将我们的域名从任何其他注册商（如 GoDaddy 等）迁移到 Route 53，或者购买 Domain AWS。

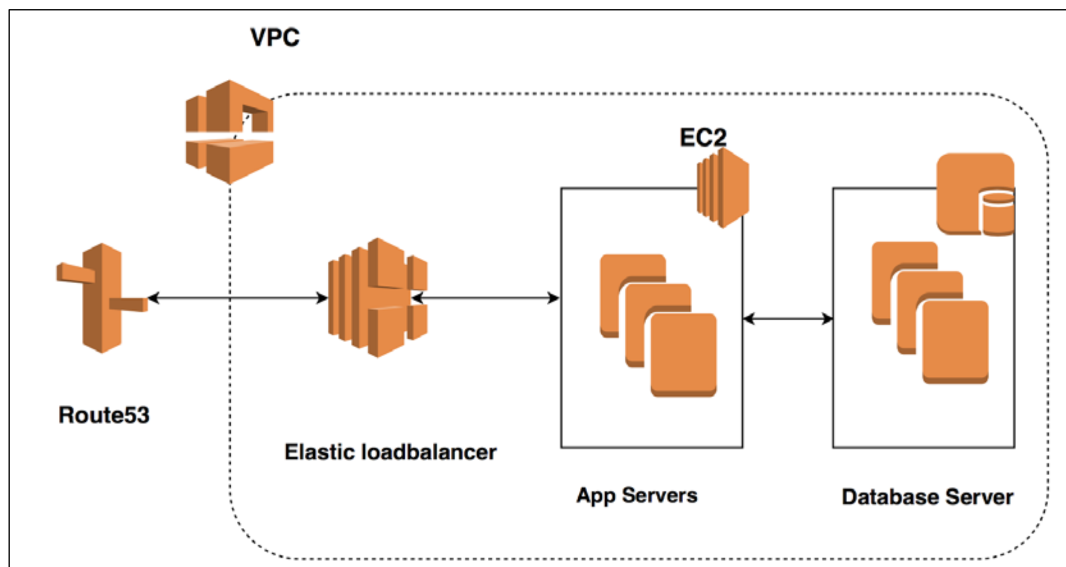
AWS 中还提供了很多其他服务，我们在此不做讨论。如果你想探索其他服务，请访问 AWS 产品列表（`https://aws.amazon.com/products/`）。

下面我们将使用上面提到的大部分服务。首先，我们在 AWS 上为每个应用程序构建基础架构。

在 AWS 上构建应用程序基础架构

应用程序到了这个阶段后，系统架构师或者 DevOps 人员将会参与进来，并且提出不同的基础设施方案，这些方案安全高效，足以处理应用程序的请求，同时也具备成本效益。

对于我们的应用，我们将构建如下图所示的基础架构。



我们将按照该架构图来构建应用程序，其中包括了一些 AWS 服务，如 EC2、VPC、Route 53 等。

在 AWS 上有三种不同的配置资源的方式。

- **管理控制台**：这是我们已经登录过的用户界面，可以使用它在云上启动资源（请参考该链接：`https://console.aws.amazon.com/console/`）。
- **以编程方式**：我们可能会使用一些编程语言（如 Python、Ruby 等）和 AWS 的其他开发工具来创建资源，如 Codecom。另外，你还可以使用你喜欢的语言的 SDK 来创建资源。参阅 `https://aws.amazon.com/tools/`了解更多信息。
- **AWS CLI (Command-line interface)**：这是一款基于 Python 的 SDK 构建的开源工具，其提供了与 AWS 资源交互的命令。可以通过 `http://docs.aws.amazon.com/cli/latest/userguide/cli-chap-welcome.html` 了解其工作方式以及在你的系统上进行设置的步骤。

在 AWS 上创建资源十分简单，所以我们不介绍这部分内容，你可以参考 AWS 的文档（`https://aws.amazon.com/documentation/`）来创建。

下面我们将介绍如何使用 Terraform 和基于 AWS 的服务 CloudFormation 构建基础架构。

生成认证密钥

身份验证是所有平台和产品中都需要有的重要功能，它用于检查尝试访问和执行产品操作的用户的真实性，并确保系统的安全。由于此处我们将使用 API 访问 AWS 账户，因此我们需要使用认证密钥来验证请求。我们现在看到的画面是 IAM（身份和访问管理）。

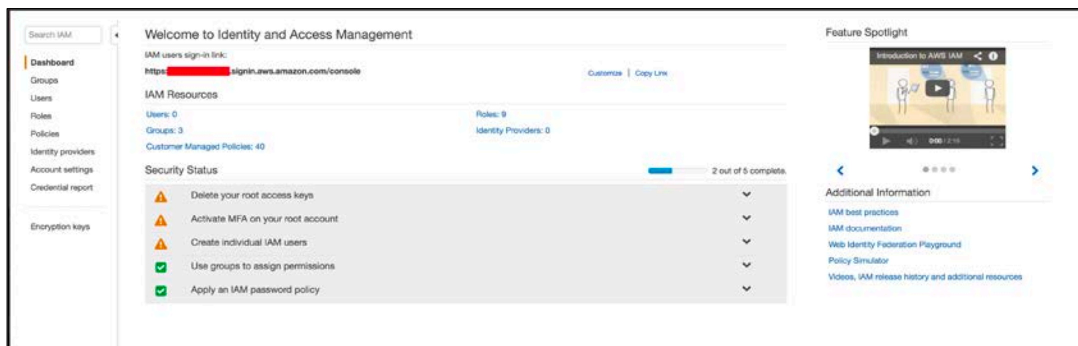
在 IAM 中，我们定义用户并生成访问/秘密密钥，且根据要使用的资源分配角色。



强烈建议不要以 root 用户身份生成访问/秘密密钥，因为默认情况下，它将拥有对你的帐户的完全访问权。

以下是创建用户及生成访问/秘密密钥的步骤。

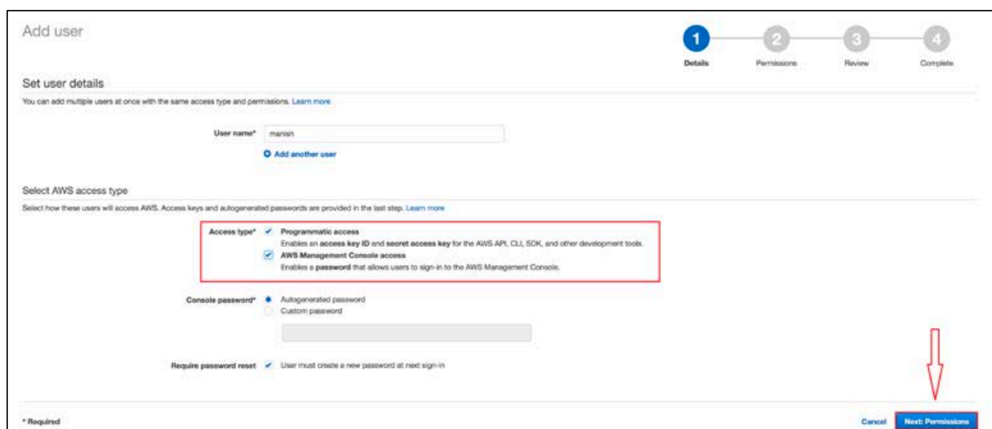
1. 访问 `https://console.aws.amazon.com/iam/home?region=us-east-1#/home`，你将看到如下图所示的界面。



2. 单击左侧面板中的第三个选项 **Users**。如果你使用的是新创建的账户，你看到的将是不同的用户。现在，我们创建一个新用户——为此，单击右侧面板中的 **Add user** 按钮，如下图所示。



3. 单击 **Add user** 按钮后，将会加载一个新的页面，询问你用户名和你想要让用户访问账户的方式。如果你将要使用此用户（例如 manish），仅出于编程目的，则建议你取消选中 **AWS Management Console access** 框，以禁止用户通过 AWS 管理控制台登录，如下图所示。



4. 完成后，单击屏幕右下角的 **Next: Permissions** 按钮。接下来，需要选择要授予此用户的权限，我们称之为 IAM 策略。这样，用户就可以根据定义的策略来访问资源，在这些策略上定义了允许用户操作的类型。现在，我们为用户添加 **Power User Access** 策略。

5. **Power User Access** 有一个 JSON 格式的策略，如下所示：

```
{
  "Version": "2012-10-17",
  "Statement": [
       {
         "Effect": "Allow",
         "NotAction": [
           "iam:*",
```

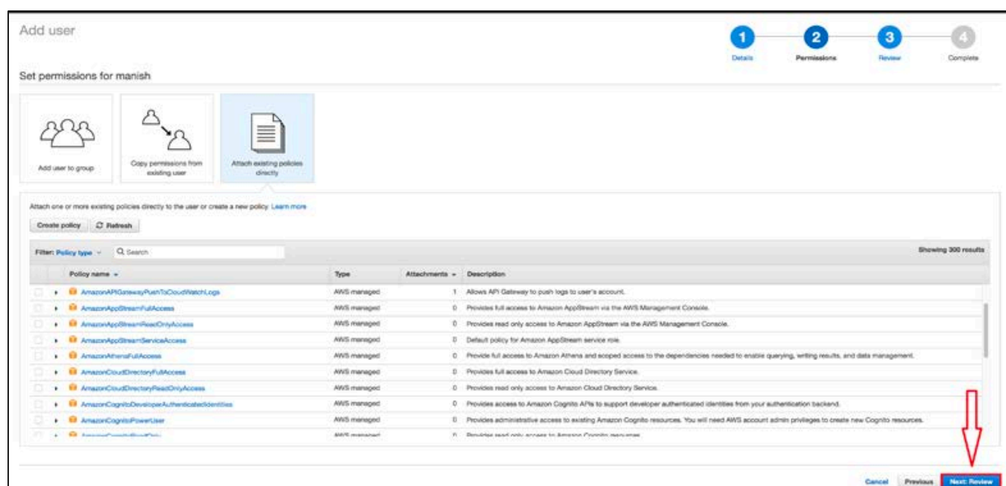
```
            "organizations:*"
          ],
          "Resource": "*"
        },
        {
          "Effect": "Allow",
            "Action": "organizations:DescribeOrganization",
            "Resource": "*"
        }
      ]
  }
```

有关 IAM 策略的更多信息，可阅读以下链接中的文档：http://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies.html。



使用 Microsoft Active Directory 的读者可以使用 AD 连接器轻松地将 AD 与 IAM 集成在一起。欲了解更多信息，请阅读此链接给出的文章：https://aws.amazon.com/blogs/security/how-to-connect-your-on-premises-active-directory-to-aws-using-ad-connector/。

在下图所示的屏幕添加策略。

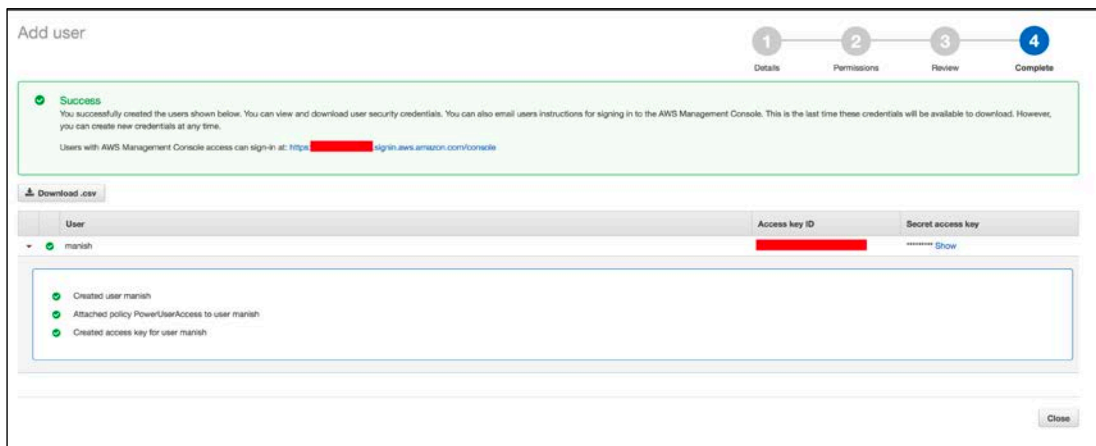


1. 将策略添加到用户后，单击屏幕右下角的 **Next: Review** 按钮以继续。

2. 下一个画面会要求你查看配置，一旦确定，你可以单击 **Create user** 按钮来创建用户。



3. 单击 **Create user** 按钮后，即会创建用户，并且把该策略附加到用户。之后将看到如下图所示的屏幕，其上有自动生成的访问密钥及秘密密钥，请确保其安全，永远不要与任何人分享。



4. 现在我们的访问/秘密密钥已经生成，可以在 AWS 上构建应用程序基础架构了。我们将使用到以下工具。

- **Terraform**：这是在不同云平台上构建基础架构的开源工具。
- **CloudFormation**：这是使用 AWS 资源构建应用程序基础架构的 AWS 服务。

Terraform——基础设施即代码构建工具

Terraform 是一款用于在不同云平台（如 AWS、Azure 等）上构建、管理和版本化基础架构的工具。它可以管理基础设施的低级组件，如计算、存储、网络等。

在 Terraform 中，我们指定了描述应用程序基础结构资源规范的配置文件。Terraform 描述了执行计划以及所要达到的状态。然后，按照规范开始构建资源，并在构建之后跟踪基础架构的当前状态，如果配置发生更改，则都采取增量式执行。

以下是 Terraform 的一些功能：

- Terraform 将你的数据中心描述为蓝图，可以对其进行版本管理并将其管理为代码。
- Terraform 在实际执行之前为你提供执行计划，帮助你将执行计划与预期结果相匹配。
- Terraform 可帮助你构建所有资源，并行化资源创建。使用它你能够深入了解资源的依赖关系，并确保在创建资源之前完成依赖关系。
- 依靠其洞察力，可以为开发人员提供更多的控制，从而减少人为错误，实现基础架构。

在 Terraform 中，我们根据需要处理的资源来看待 AWS 中的每个服务，所以需要为其创建提供强制属性。下面，我们开始创建资源。

1. 首先，需要创建 **VPC**（**Virtual Private Cloud**），我们将用它来启动所有其他资源。



按照惯例创建后缀名为.tf 的文件。

2. 所以，我们创建一个空的 main.tf 文件。添加下面的代码，设置服务提供者的访问/秘密密钥：

```
# Specify the provider and access details
    provider "aws" {
      region = "${var.aws_region}"
      access_key = "${var.aws_access_key}"
      secret_key = "${var.aws_secret_key}"
}
```

3. 上面代码中有一个`${var.aws_region}`样的值，这是为了遵循将所有值保存在一个名为 `variables.tf` 的单独文件中的惯例，所以这里要执行此操作。用下面的内容来更改 `variables.tf` 文件：

```
variable "aws_access_key" {
     description = "AWS access key"
     default = ""                          # Access key
 }
variable "aws_secret_key" {
     description = "AWS secret access key"
     default = ""                              # Secret key
 }
variable "aws_region" {
     description = "AWS region to launch servers."
     default = "us-east-1"
}
```

4. 接下来，需要创建 VPC 资源，将下面代码添加到 `main.tf` 中：

```
# Create a VPC to launch our instances into
  resource "aws_vpc" "default" {
    cidr_block = "${var.vpc_cidr}"
    enable_dns_hostnames = true
    tags {
      Name = "ms-cloud-native-app"
    }
}
```

5. 我们需要用到一个变量，在 `variables.tf` 中定义这个变量，如下所示：

```
variable "vpc_cidr"{
   default = "10.127.0.0/16"           # user defined
}
```

6. 定义了 VPC 资源后，需要创建一个用来连接 EC2 机器的子网、Elastic Load Balancer 和其他资源。所以，将下列代码添加到 `main.tf` 中：

```
# Create a subnet to launch our instances into
resource "aws_subnet" "default" {
```

```
  vpc_id               = "${aws_vpc.default.id}"
  cidr_block            = "${var.subnet_cidr}"
    map_public_ip_on_launch = true
  }
Now, define the variable we have used in above code in
variables.tf
variable "subnet_cidr"{
 default = "10.127.0.0/24"
}
```

7. 因为我们希望我们的资源可以通过互联网访问，所以需要创建一个互联网网关，并将其与我们的子网相关联，从而使在其内部创建的资源可以通过互联网访问。



可以创建多个子网来保护我们的资源。

8. 将下列代码添加到 main.tf 中：

```
# Create an internet gateway to give our subnet access to the
outside world
resource "aws_internet_gateway" "default" {
vpc_id = "${aws_vpc.default.id}"
 }
# Grant the VPC internet access on its main route table
resource "aws_route" "internet_access" {
route_table_id          = "${aws_vpc.default.main_route_table_id}"
destination_cidr_block  = "0.0.0.0/0"
gateway_id              = "${aws_internet_gateway.default.id}"
```

9. 接下来，需要确保启动的 EC2 机器拥有子网地址。在 main.tf 中添加如下代码：

```
# Create a subnet to launch our instances into
resource "aws_subnet" "default" {
  vpc_id                     = "${aws_vpc.default.id}"
  cidr_block                 = "${var.subnet_cidr}"
  map_public_ip_on_launch   = true
}
```

10. 配置完成后，可以创建 app server 和 MongoDB server。

11. 需要在一开始的时候就创建依赖资源，如安全组，如果没有它，就无法启动 EC2。

12. 向 `main.tf` 中添加以下代码来创建安全组资源：

```
# the instances over SSH and HTTP
resource "aws_security_group" "default" {
name            = "cna-sg-ec2"
description     = "Security group of app servers"
vpc_id          = "${aws_vpc.default.id}"
# SSH access from anywhere
ingress {
 from_port      = 22
 to_port        = 22
 protocol       = "tcp"
 cidr_blocks    = ["0.0.0.0/0"]
}
# HTTP access from the VPC
ingress {
  from_port     = 22
  to_port       = 22
  protocol      = "tcp"
  cidr_blocks   = ["${var.vpc_cidr}"]
 }
 # outbound internet access
 egress {
  from_port     = 0
  to_port       = 0
  protocol      = "-1"
  cidr_blocks   = ["0.0.0.0/0"]
 }
}
```

13. 我们在安全组中开放了 `22` 端口和 `5000` 端口，用来 ssh 和访问应用程序。

14. 接下来，需要添加/创建 ssh 密钥对，可以在本地计算机上生成 ssh 密钥对并将其上传到 AWS，或者也可以从 AWS 控制台生成 ssh 密钥对。在我们的例子中，使用

ssh-keygen 命令在本地机器上生成了一个 ssh 密钥。现在要在 AWS 中创建一个 ssh-key 密钥对资源，将下面代码添加到 main.tf 中：

```
resource "aws_key_pair" "auth" {
  key_name = "${var.key_name}"
   public_key = "${file(var.public_key_path)}"
}
```

15. 将下面的代码片段添加到 variables.tf 中以向变量提供参数：

```
variable "public_key_path" {
  default = "ms-cna.pub"
}
```

16. 现在我们已经创建了依赖资源，可以创建应用程序服务器（即 EC2 机器）了。因此，将下面的代码片段添加到 main.tf 中：

```
resource "aws_instance" "web" {
 # The connection block tells our provisioner how to
 # communicate with the resource (instance)
  connection {
   # The default username for our AMI
    user = "ubuntu"
    key_file = "${var.key_file_path}"
    timeout = "5m"
  }
 # Tags for machine
 tags {Name = "cna-web"}
 instance_type = "t2.micro"
 # Number of EC2 to spin up
  count = "1"
  ami = "${lookup(var.aws_amis, var.aws_region)}"
  iam_instance_profile = "CodeDeploy-Instance-Role"
  # The name of our SSH keypair we created above.
  key_name = "${aws_key_pair.auth.id}"
 # Our Security group to allow HTTP and SSH access
 vpc_security_group_ids = ["${aws_security_group.default.id}"]
 subnet_id = "${aws_subnet.default.id}"
```

```
}
```

17. 在 EC2 配置中也使用了一些变量，所以需要在 `variables.tf` 文件中添加变量值：

```
 variable "key_name" {
  description = "Desired name of AWS key pair"
  default = "ms-cna"
 }
variable "key_file_path" {
   description = "Private Key Location"
   default = "~/.ssh/ms-cna"
}
 # Ubuntu Precise 12.04 LTS (x64)
  variable "aws_amis" {
    default = {
    eu-west-1 = "ami-b1cf19c6"
    us-east-1 = "ami-0a92db1d"
    #us-east-1 = "ami-e881c6ff"
    us-west-1 = "ami-3f75767a"
    us-west-2 = "ami-21f78e11"
  }
}
```

现在，app server 的资源配置已经完成了。接下来，为了保存数据，需要为 MongoDB 服务器添加一个类似的配置。准备就绪后，我们创建 ELB（这将是用户的应用程序访问点），然后将 app server 附加到 ELB。

接下来添加 MongoDB 服务器配置。

配置 MongoDB 服务器

将以下代码添加到 `main.tf` 中以创建 MongoDB 服务器的安全组：

```
resource "aws_security_group" "mongodb" {
 name        = "cna-sg-mongodb"
 description = "Security group of mongodb server"
 vpc_id      = "${aws_vpc.default.id}"
```

```
# SSH access from anywhere
ingress {
  from_port = 22
  to_port = 22
  protocol = "tcp"
  cidr_blocks = ["0.0.0.0/0"]
 }
# HTTP access from the VPC
ingress {
  from_port = 27017
  to_port = 27017
  protocol = "tcp"
  cidr_blocks = ["${var.vpc_cidr}"]
 }
# HTTP access from the VPC
ingress {
  from_port = 27017
  to_port = 27017
  protocol = "tcp"
  cidr_blocks = ["${var.vpc_cidr}"]
  }
# outbound internet access
egress {
  from_port = 0
  to_port = 0
  protocol = "-1"
  cidr_blocks = ["0.0.0.0/0"]
 }
}
```

接下来，需要添加 MongoDB 服务器配置。另请注意，在以下配置中，我们在创建的 EC2 机器上安装 MongoDB：

```
resource "aws_instance" "mongodb" {
# The connection block tells our provisioner how to
# communicate with the resource (instance)
connection {
```

```
    # The default username for our AMI
    user = "ubuntu"
    private_key = "${file(var.key_file_path)}"
    timeout = "5m"
    # The connection will use the local SSH agent for authentication.
    }
   # Tags for machine
   tags {Name = "cna-web-mongodb"}
   instance_type = "t2.micro"
   # Number of EC2 to spin up
   count = "1"
   # Lookup the correct AMI based on the region
   # we specified
   ami = "${lookup(var.aws_amis, var.aws_region)}"
   iam_instance_profile = "CodeDeploy-Instance-Role"
   # The name of our SSH keypair we created above.
    key_name = "${aws_key_pair.auth.id}"
    # Our Security group to allow HTTP and SSH access
    vpc_security_group_ids = ["${aws_security_group.mongodb.id}"]
    subnet_id = "${aws_subnet.default.id}"
    provisioner "remote-exec" {
     inline = [
       "sudo echo -ne '\n' | apt-key adv --keyserver
        hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10",
       "echo 'deb http://repo.mongodb.org/apt/ubuntu trusty/mongodb-
        org/3.2 multiverse' | sudo tee /etc/apt/sources.list.d/mongodb-
         org-3.2.list",
       "sudo apt-get update -y && sudo apt-get install mongodb-org --
       force-yes -y",
       ]
     }
   }
```

最后还需要配置一个 Elastic Load Balancer 资源，用于对客户的请求进行负载均衡以提供高可用性。

配置 Elastic Load Balancer

首先，需要为 ELB 创建一个安全组资源，为此，向 main.tf 中添加如下代码：

```
# A security group for the ELB so it is accessible via the web
 resource "aws_security_group" "elb" {
 name        = "cna_sg_elb"
 description = "Security_group_elb"
 vpc_id      = "${aws_vpc.default.id}"
# HTTP access from anywhere
ingress {
  from_port = 5000
  to_port = 5000
  protocol = "tcp"
  cidr_blocks = ["0.0.0.0/0"]
 }
# outbound internet access
egress {
  from_port = 5000
  to_port = 5000
  protocol = "tcp"
  cidr_blocks = ["0.0.0.0/0"]
}
```

然后，通过添加如下配置来创建 ELB 资源，同时添加 app server：

```
resource "aws_elb" "web" {
name = "cna-elb"
 subnets        = ["${aws_subnet.default.id}"]
 security_groups = ["${aws_security_group.elb.id}"]
 instances      = ["${aws_instance.web.*.id}"]
 listener {
   instance_port = 5000
   instance_protocol = "http"
   lb_port = 80
   lb_protocol = "http"
  }
}
```

至此，我们的第一个 Terraform 配置就完成了。

我们的基础架构配置已经完成，下面准备部署。使用以下命令来了解执行计划：

```
$ terraform plan
```

最后一个命令的输出应该如下图所示。

```
root@packtpub:/vagrant/github/Cloud-Native-Python/chapter 11/terraform-app# terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.

The Terraform execution plan has been generated and is shown below.
Resources are shown in alphabetical order for quick scanning. Green resources
will be created (or destroyed and then created if an existing resource
exists), yellow resources are being changed in-place, and red resources
will be destroyed. Cyan entries are data sources to be read.

Note: You didn't specify an "-out" parameter to save this plan, so when
"apply" is called, Terraform can't guarantee this is what will execute.

+ aws_elb.web
    availability_zones.#:                   "<computed>"
    connection_draining:                    "false"
    connection_draining_timeout:            "300"
    cross_zone_load_balancing:              "true"
    dns_name:                               "<computed>"
    health_check.#:                         "<computed>"
    idle_timeout:                           "60"
    instances.#:                            "<computed>"
    internal:                               "<computed>"
    listener.#:                             "1"
    listener.996561874.instance_port:       "5000"
    listener.996561874.instance_protocol:   "http"
    listener.996561874.lb_port:             "80"
    listener.996561874.lb_protocol:         "http"
    listener.996561874.ssl_certificate_id:  ""
    name:                                   "cna-elb"
    security_groups.#:                      "<computed>"
    source_security_group:                  "<computed>"
    source_security_group_id:               "<computed>"
    subnets.#:                              "<computed>"
```

如果没有看到任何错误，则可以执行以下命令来实际创建资源：

```
$ terraform apply
```

输出应该如下图所示。

```
root@packtpub:/vagrant/github/Cloud-Native-Python/chapter 11/terraform-app# terraform apply
aws_vpc.default: Creating...
  assign_generated_ipv6_cidr_block: "" => "false"
  cidr_block:                       "" => "10.127.0.0/16"
  default_network_acl_id:           "" => "<computed>"
  default_route_table_id:           "" => "<computed>"
  default_security_group_id:        "" => "<computed>"
  dhcp_options_id:                  "" => "<computed>"
  enable_classiclink:               "" => "<computed>"
  enable_dns_hostnames:             "" => "true"
  enable_dns_support:               "" => "true"
  instance_tenancy:                 "" => "<computed>"
  ipv6_association_id:              "" => "<computed>"
  ipv6_cidr_block:                  "" => "<computed>"
  main_route_table_id:              "" => "<computed>"
  tags.%:                           "" => "1"
  tags.Name:                        "" => "ms-cloud-native-app"
aws_key_pair.auth: Creating...
  fingerprint: "" => "<computed>"
  key_name:    "" => "ms-cna"
  public_key:  "" => "ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDUSpeFPkaru+J3l5MXWlVUJp/vFC+IJKCtrj9R+FInUIAE6qxJ5TffUKSHHPODW4Xa9s6V0QOShp/HtzI1yBar
LrDgbDMN+laXlIy3vdBP42WxvEDUIuGTAepRDWpun5ANeVnTPpVRb9eWQtkaKSb1cB+BXN3t0qAWY9bwFyT5ijqyXmMg16W4ihgRR5trlcdhpTTc6DG310uFmChDjxN+vArRGcduIJYMOgFJaF
JDJDtjjJ5mDq4JHza89I4ziCN6sv6g7eq8stVDXdc47w7Sn4unySWD8kG2nHO2Xjl8YPAHCEwJapgu/tTB/sXReeKEDh8ZKBrXJGpkkAB7/ZtV root@packtpub"
aws_vpc.default: Still creating... (10s elapsed)
aws_key_pair.auth: Still creating... (10s elapsed)
aws_vpc.default: Still creating... (20s elapsed)
aws_key_pair.auth: Still creating... (20s elapsed)
aws_vpc.default: Still creating... (30s elapsed)
aws_key_pair.auth: Still creating... (30s elapsed)
aws_key_pair.auth: Creation complete (ID: ms-cna)
aws_vpc.default: Still creating... (40s elapsed)
aws_vpc.default: Still creating... (50s elapsed)
aws_vpc.default: Still creating... (1m0s elapsed)
aws_vpc.default: Still creating... (1m10s elapsed)
aws_vpc.default: Still creating... (1m20s elapsed)
aws_vpc.default: Still creating... (1m30s elapsed)
aws_vpc.default: Still creating... (1m40s elapsed)
aws_vpc.default: Still creating... (1m50s elapsed)
```

目前,我们没有注册域名,如果我们在 Route 53 中注册和配置域名,则需要在 main.tf 中创建一个额外的资源，为我们的应用添加一个条目。可以使用下面的代码来注册：

```
resource "aws_route53_record" "www" {
  zone_id = "${var.zone_id}"
  name = "www.domain.com"
  type = "A"
  alias {
   name = "${aws_elb.web.dns_name}"
   zone_id = "${aws_elb.web.zone_id}"
   evaluate_target_health = true
  }
}
```

仅此而已。而且，还有一个能够使我们的基础设施高可用的快捷方式是基于服务器使用率指标（CPU 或内存），创建自动伸缩服务。我们设置了服务器扩展或者收缩的条件以减少应用程序的延迟。



为此，请参阅 Terraform 的文档：`https://www.terraform.io/docs/providers/aws/r/autoscaling_group.html`

到目前为止，还没有部署应用程序，我们将使用 CodeDeploy 持续交付服务来部署应用程序，这将在本章的后面讨论。

在此之前,我们来看看如何使用由 AWS 提供的名为 **CloudFormation** 的 Cloud Platform Service 创建相同的设置。

CloudFormation——构建基础设施即代码的 AWS 工具

CloudFormation 是一个 AWS 服务，其工作方式与 Terraform 相似。但是，在 CloudFormation 中，我们不需要访问/秘密密钥。相反，只需要创建一个 IAM 角色，该角色将具有启动构建应用程序所需的所有资源的必要权限。

可以使用 YAML 或者 JSON 格式编写 CloudFormation 配置。

下面我们使用 CloudFormation 构建 VPC，该 VPC 将包含一个公共的子网和一个私有的子网。

创建一个新的配置文件 `vpc.template`，配置 VPC 和子网（公共和私有的）如下：

```
"Resources" : {
   "VPC" : {
     "Type" : "AWS::EC2::VPC",
     "Properties" : {
       "CidrBlock" : "172.31.0.0/16",
       "Tags" : [
         {"Key" : "Application", "Value" : { "Ref" : "AWS::StackName"} },
         {"Key" : "Network", "Value" : "Public" }
       ]
     }
   },
"PublicSubnet" : {
     "Type" : "AWS::EC2::Subnet",
     "Properties" : {
       "VpcId" : { "Ref" : "VPC" },
```

```
        "CidrBlock" : "172.31.16.0/20",
        "AvailabilityZone" : { "Fn::Select": [ "0", {"Fn::GetAZs": {"Ref":
"AWS::Region"}} ]},
        "Tags" : [
          {"Key" : "Application", "Value" : { "Ref" : "AWS::StackName"} },
          {"Key" : "Network", "Value" : "Public" }
        ]
      }
    },
    "PrivateSubnet" : {
      "Type" : "AWS::EC2::Subnet",
      "Properties" : {
        "VpcId" : { "Ref" : "VPC" },
        "CidrBlock" : "172.31.0.0/20",
        "AvailabilityZone" : { "Fn::Select": [ "0", {"Fn::GetAZs": {"Ref":
"AWS::Region"}} ]},
        "Tags" : [
          {"Key" : "Application", "Value" : { "Ref" : "AWS::StackName"} },
          {"Key" : "Network", "Value" : "Public" }
        ]
    }
},
```

该文件使用 JSON 格式。此外，我们还指定了路由表和 internet 网关的配置，如下所示：

```
"PublicRouteTable" : {
      "Type" : "AWS::EC2::RouteTable",
      "Properties" : {
        "VpcId" : {"Ref" : "VPC"},
        "Tags" : [
          {"Key" : "Application", "Value" : { "Ref" : "AWS::StackName"} },
          {"Key" : "Network", "Value" : "Public" }
        ]
      }
},
    "PublicRoute" : {
      "Type" : "AWS::EC2::Route",
      "Properties" : {
```

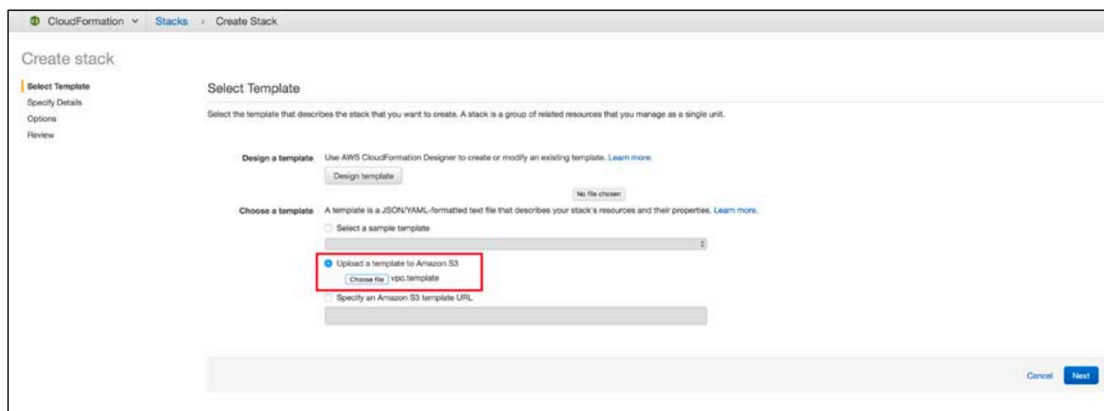
```
          "RouteTableId" : { "Ref" : "PublicRouteTable" },
          "DestinationCidrBlock" : "0.0.0.0/0",
          "GatewayId" : { "Ref" : "InternetGateway" }
        }
      },
      "PublicSubnetRouteTableAssociation" : {
        "Type" : "AWS::EC2::SubnetRouteTableAssociation",
        "Properties" : {
          "SubnetId" : { "Ref" : "PublicSubnet" },
          "RouteTableId" : { "Ref" : "PublicRouteTable" }
        }
      }
   },
```

准备好配置文件后，我们使用 AWS 控制台来创建 VPC 栈。

AWS 上的 VPC 栈

执行下面步骤，使用 AWS 控制台创建 VPC 栈：

1. 访问 `https://console.aws.amazon.com/cloudformation/home? region=us-east-1#/stacks/new`，使用 CloudFormation 创建一个新的栈。你将看到如下图所示的画面。



提供模板文件的路径，然后单击 **Next** 按钮。

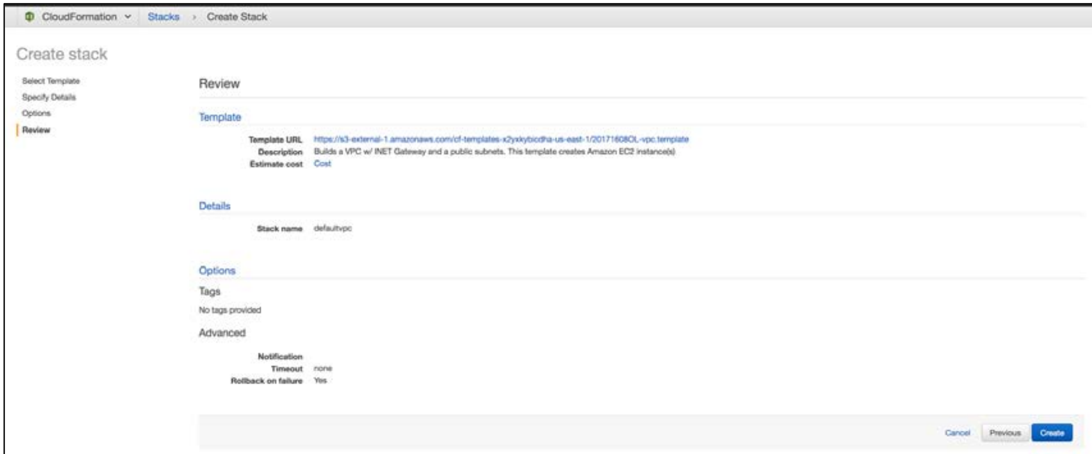
2. 在下一个窗口中，指定 **Stack name** 作为栈的唯一标识，如下图所示。

提供栈的名称后，单击 **Next** 按钮。

3. 下一个界面是可选的；如果你要设置 SNS（通知服务），或者为它添加 IAM 角色，则需要在这里添加设置，如下图所示。

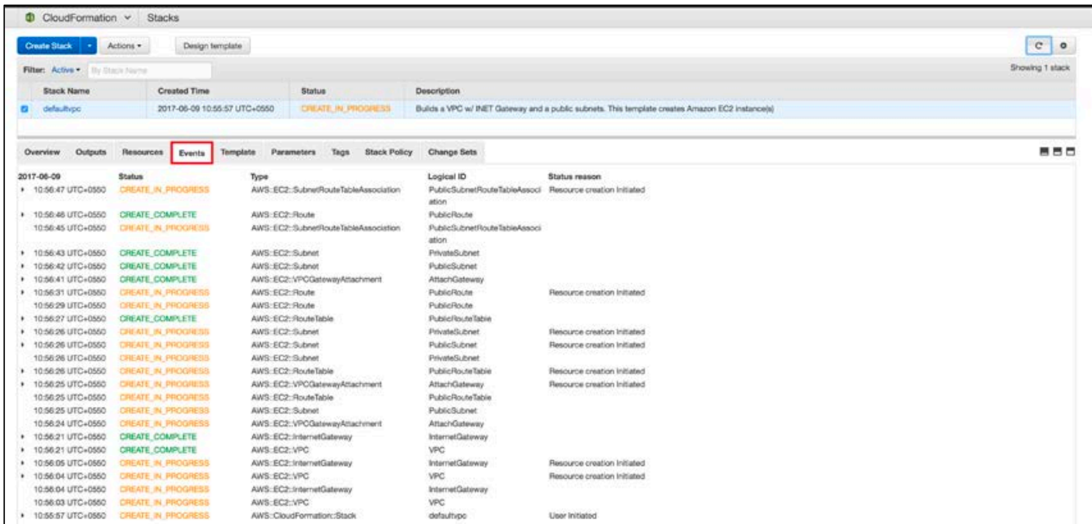
如果你要启用通知服务和 IAM 角色，则添加详细信息后，单击 **Next** 按钮。

4. 下一个界面用于查看详细信息，确保创建的栈是正确的，如下图所示。



准备就绪后，单击 **Create** 按钮开始创建栈。在创建时，你可以检查事件以了解资源创建的状态。

你将看到如下图所示的画面。



在该画面中，你将看到整个栈的构建进度，一旦发生错误，就可以通过事件来调试。

VPC 栈构建好了之后，在该 VPC 中创建 EC2、ELB 和自动扩展资源。我们将给出 YAML 格式的配置说明。

你可以在<path of repository>中找到完整的代码。我们将使用 `main.yam` 文件，其中包含了示例所在的 VPC 和子网的详细配置。

5. 启动该栈，参考以下链接中的说明：`https://console.aws.amazon.com/cloudformation/home?region=us-east-1#/stacks/new`。

在启动配置中需要做一点改动——不是在文件中指定值，而是在 AWS 控制台中进行详细配置时指定，如下图所示。

6. 参考下图，配置你要部署的应用程序实例的详细信息。

Instance Settings

AMI ami-80861296 Enter Baked AMI ID.

OS Type ubuntu Select OS Type of the AMI.

Instance Type t2.micro Select Instance Type.

EC2 Keypair ms-cna
Select Keypair to Assign to EC2 Instances

Instance HTTP Port 5000 Enter HTTP Listening Port for Instance.

Instance-to-Instance SG Search by ID, name or Name tag value
Security Group That Allows Bastion Host Access to Instances.

Remote-to-Instance SG Search by ID, name or Name tag value
Remote Network or IP that can Access the instances of VPN or Direct Connect.

Network Configuration

VPC Search by ID, or Name tag value
Select VPC.

Public Subnet 1 Search by ID, or Name tag value
Public Subnet 1 to Deploy ELB to.

Private Subnet 1 Search by ID, or Name tag value
Private Subnet 1 to Deploy app Autoscaling Group to.

Route 53 DNS Configuration

Configure DNS false Configure Route 53 DNS to ELB? Be sure a matching record doesn't already exist.

Hosted Zone domain.com (Skip if Not Configuring Route 53) Hosted DNS Name.

ELB DNS Alias www.domain.com (Skip if Not Configuring Route 53) DNS Record to Update for the ELB.

7. 配置了所有详细信息后，单击页面底部的 **Next** 按钮，跳转到 ELB 配置页面，如下图所示。

其余步骤与创建 AWS CloudFormation 堆栈一样。在 `main.yaml` 中添加 EC2 配置来添加 MongoDB 服务器。

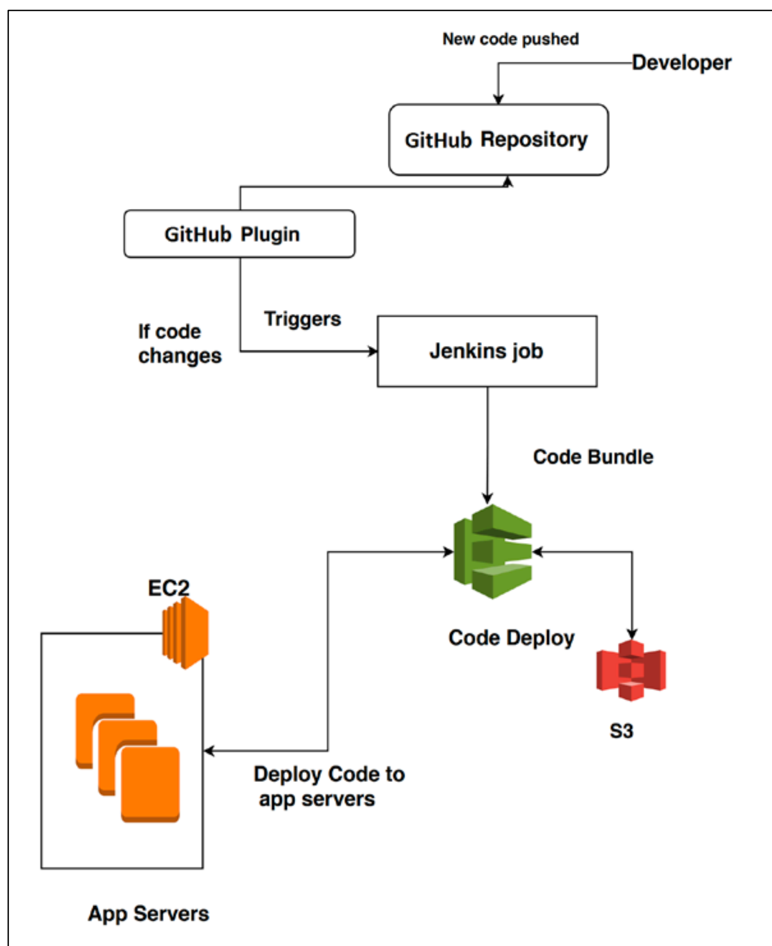


使用 AWS CloudFormation 创建配置非常简单，因为 AWS 已经提供了一些模板，我们只需要修改这些模板即可。以下是模板所在的链接：`https://aws.amazon.com/cloudformation/aws-cloudformation-templates/`。

现在基础设施已经构建完毕，可以部署我们的应用程序了。

云原生应用的持续部署

在上一节中，我们构建了基础设施，但是还没有部署应用程序。现在我们需要确定如何进行持续部署（Continuous Deployment）。因为我们在本地机器上有开发环境，所以不需要配置持续集成。但是，对于拥有众多互相协同的开发人员的大公司来说，可能需要使用像 Jenkins 这样的工具来构建一个持续集成（Continous Integration）的流水线。对于我们的示例，只需要持续部署。我们的持续部署流水线如下图所示。



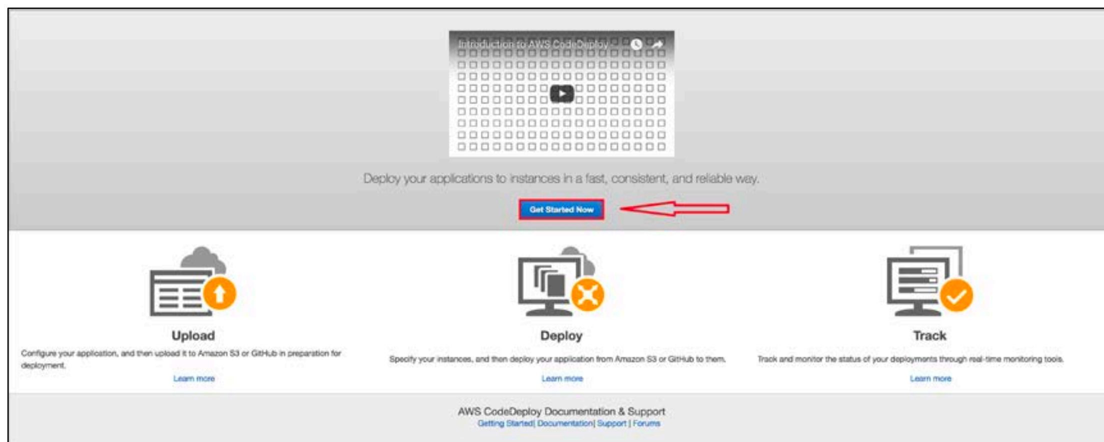
工作原理

从开发人员将代码推送到版本控制系统（在我们的例子中是 GitHub）的主分支开始，Jenkins 的 **GitHub** 插件检测到定义作业中的更改，触发 **Jenkins 作业**将新代码部署到基础设施中。然后 Jenkins 与 **Code Deploy** 通信，触发代码部署到 Amazon EC2 上。由于我们需要确保部署成功，所以需要设置一个状态通知，以便在部署失败的时候检查和恢复。

实现持续部署流水线

首先配置 AWS 服务，从 CodeDeploy 开始，将应用程序部署到可用的服务器上。

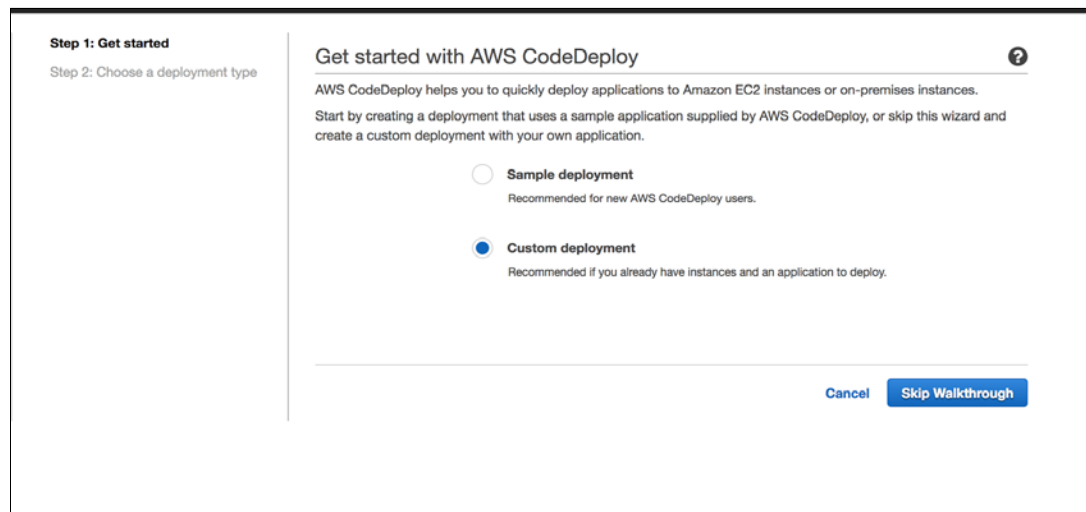
1. 当你首次访问 CodeDeploy（`https://us-west-1.console.aws. amazon.com/codedeploy/`）时，你会看到如下所示的画面。



这是 CodeDeploy 的介绍页面。

2. 单击页面中间的 **Get Started Now** 按钮以继续。

3. 接下来，你将看到如下图所示的画面，推荐你部署一个示例应用程序，这对于新手很有用。但是我们已经建立了基础设施，此时，我们需要选择自定义部署——这将跳过指导阶段。选择该选项，然后单击 **Next** 按钮。



4. 单击 **Skip Walkthrough** 按钮以继续。

5. 在下一个向导界面中，有几个部分需要检查一下。

第一个部分是要求你创建应用程序——需要提供自定义的应用程序和部署组的名称，这是必需的，因为这将会作为应用程序的标识符，如下图所示。

AWS CodeDeploy

Create application

Create an application and choose a deployment type. Specify the instances to deploy to. Specify the conditions for a successful deployment.

Application name* cloud-native-app

Deployment group name* prod

6. 滚动到下一个部分，这是关于应用程序部署类型的。有两种类型，定义如下（如下图所示）。

- **Blue/green deployment（蓝/绿部署）**：在这种类型的部署中，启动新实例，并部署新代码，如果运行状况检查正常，则替换旧实例，然后终止旧实例。建议在客户负担不起停机时间的生产环境中使用该选项。
- **In-place deployment（就地部署）**：在此部署类型中，新代码被直接部署到现有实例中，而且每个实例都将脱机更新。

  我们将采用就地部署，但是部署类型的选择依场景和产品所有者而决定。比如说，像 Uber 或者 Facebook 这样的应用程序，在部署的时候不能承受停机时间，这样的应用程序就应该进行蓝/绿部署，这将给它们提供高可用性。

Deployment type

Choose the deployment to use to deploy your application. Learn more

In-place deployment

Updates the instances in the deployment group with the latest application revision. During a deployment, each instance will be briefly taken offline for its update.

Blue/green deployment

Replaces the instances in the deployment group with new instances and deploys the latest application revision to them. After instances in the replacement environment are registered with a load balancer, instances from the original environment are deregistered and can be terminated.

7. 接下来考虑应用程序部署的基础结构。指定实例和 ELB 的细节，如下图所示。

Add instances

Identify the instances you want to include in the deployment group. We will deploy the application revision to the instances that match the instance tag keys and values or Auto Scaling group names you specify.

Requirements for each instance in the deployment:

1. Each Amazon EC2 instance must be launched with the correct IAM instance profile attached. Learn more
2. Each Amazon EC2 instance must have identifying Amazon EC2 tags (Learn more) or be in an Auto Scaling group. Learn more
3. Each on-premises instance must have an associated IAM user, identifying on-premises instance tags, and a configuration file. Learn more
4. The AWS CodeDeploy agent must be installed and running on each instance. Learn more

Search by tags

	Tag type	Key	Value	Instances
1	Amazon EC2	Name	cna-web	1
2	Amazon EC2			

Total matching instances: 1 　　1 to 1 of 1 instances

Instance ID	Status	Instance type
i-0856c4a4d8aae5d24	Running	Amazon EC2

Load balancer　Select a load balancer to manage incoming traffic during the deployment process. The load balancer blocks traffic from each instance while it's being deployed to and allows traffic to it again after the deployment succeeds.

cna-elb

8. 在下一部分中，我们将定义部署应用程序的方式。例如，假设你有 10 个实例，你可能希望一次部署全部实例，或者一次部署一个，或者一次部署总实例数的一半。我们将使用默认选项，即 **CodeDeployDefault.OneAtTime**，如下图所示。

Deployment configuration

Choose from a list of default and custom deployment configurations. A deployment configuration is a set of rules that determines how fast an application will be deployed and the success or failure conditions for a deployment.

Deployment configuration* CodeDeployDefault.OneAtATime

Deploys to one instance at a time. Succeeds if all instances or all but the last instance succeeds. Fails after any instance except the last instance fails. Allows the deployment to succeed for some instances, even if the overall deployment fails.

Advanced

Service role

Select a service role that grants AWS CodeDeploy access to the instances.

Service role ARN*

*Required

Cancel Create application

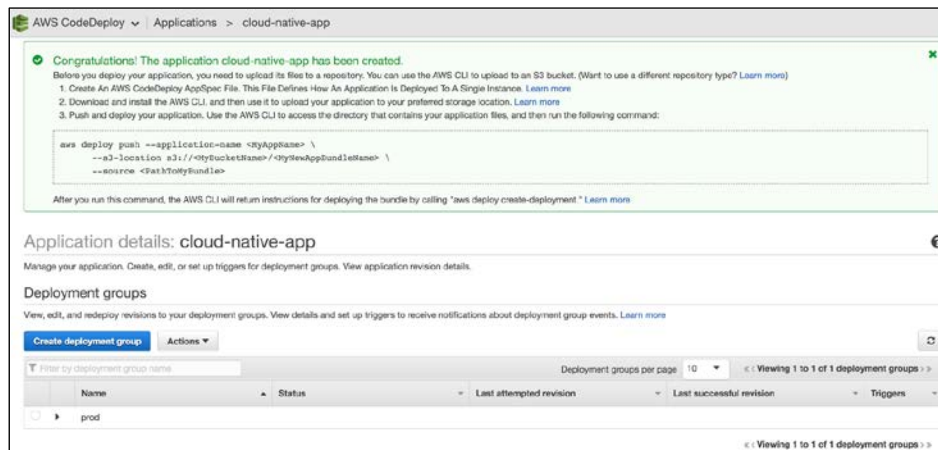
在这里，指定 CodeDeploy 需要的**服务角色**，以在你的 AWS 资源（更具体地说是在 EC2 和 ELB 上）上执行操作。



要了解更多关于服务角色创建的信息，请参考以下链接中的 AWS 文档：`http://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles_create.html`。

9.　提供所需的信息之后，单击 **Create Application** 按钮。

应用程序部署完成后，你将看到如下图所示的画面。

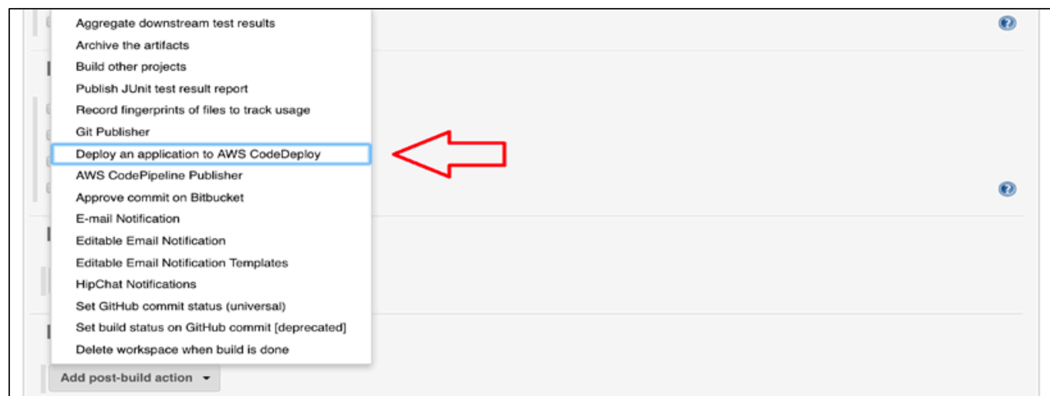


现在已经准备好部署了。我们还需要在 Jenkins 中创建一个 job，并添加一个 CodeDeploy 的 post-build。

创建 job 的流程跟我们在前一章中讲的类似，但是有几个地方需要更改。

1. 首先，需要确保已经安装了一些 Jenkins 插件，即 Jenkins 的 AWS CodeDeploy 插件、Git 插件、GitHub 插件，等等。

2.　如果已经安装了这些插件，你应该可以在 post-build 动作列表中看到下面图中所示的动作。



3. 接下来，你需要选择 **Deploy an application to AWS CodeDeploy** 动作。之后画面中将添加一个新的部分，显示在 AWS 控制台中创建的 CodeDeploy 应用程序的详细信息，如下图所示。

4. 还需要提供我们在本章开头创建的访问/秘密密钥。这是必要的，因为 Jenkins 在打包应用程序之后需要将其上传到 S3，并指示 CodeDeploy 从指定存储桶部署最新版本的代码。

现在我们的 Jenkins 流水线已经配置完成了，可以开始部署应用程序了！赶快试一下吧！

本章小结

这一章从某种程度上说非常有趣。首先，我们了解了 AWS 服务，知道了如何充分利用 AWS 服务。接下来，探讨了 AWS 上的应用程序体系结构，这将使你形成构建不同应用程序/产品架构的观念。我们还使用 Terraform 这个第三方工具，在 AWS 上构建基础架构代码。最后，部署了我们的应用程序，并使用 Jenkins 创建了持续部署流水线。在下一章中，我们将探讨微软 Azure 云平台。你需要保持活力，准备探索 Azure。下一章见！

12

部署到 Azure 云平台

在上一章中，我们了解了用于托管应用程序的平台——AWS，它包含让应用程序高可用、无停机时间等所有功能特性。在本章中，我们将介绍另一款云平台——**Microsoft Azure**。

本章包含以下主题：

- Microsoft Azure
- 构建用于应用程序基础架构的 Azure 服务
- 在 Azure 中使用 Jenkins CI/CD

Microsoft Azure 入门

顾名思义，Microsoft Azure 是微软提供的一个公共云平台，可以为客户提供不同的 PaaS 和 IaaS 服务，包括流行的虚拟机服务、应用程序服务、SQL 数据库和资源管理器等。

Azure 服务主要分为以下两大类。

- **平台服务**：这些服务可以为用户提供构建、管理和运行应用程序的环境，同时自行处理基础架构。下面是 Azure 服务的分类。
  - — 管理服务：这些服务提供管理门户和市场服务，为 Azure 中的自动化提供仓库和工具。
  - — 计算：包含诸如 fabric、函数等服务，这些服务可以帮助开发者开发和部署高可扩展的应用程序。
  - — CDN 和媒体服务：它们分别在全球范围内提供安全可靠的内容分发和实时流

式传输服务。

— Web 和移动端：这些服务用于构建 Web 应用和 API 应用，主要用于构建 Web 和移动应用程序。

— 分析：这些是与大数据相关的服务，可帮助机器学习开发人员执行实时数据处理，使你深入了解诸如 HDInsight、机器学习、流分析、Bot 服务等数据。

— 开发工具：这些服务用于版本控制、协作等，其中还包含 SDK。

— 人工智能和认知服务：这些是基于人工智能的服务，如语音、视觉等。文本分析 API、认知应用可能会用到这些服务。

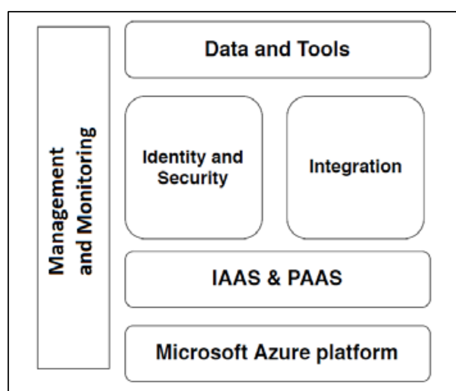
- **基础设施服务**：服务提供商会负责硬件的维护，而由客户来自定义硬件和其规格。

— 服务器计算和容器：如虚拟机和容器，这些服务为用户的计算能力提供了不同的载体。

— 存储：包括两种类型——BLOB 和文件存储。它们根据延迟和速度而具有不同的用途。

— 网络：提供与网络相关的服务，如负载均衡器和虚拟网络，其可以用来保护你的网络，并对客户做出最佳网络响应。

下图将有助于你了解 Azure 平台。



你可以在下面的链接里查看 Microsoft Azure 提供的所有产品的信息：

```
https://azure.microsoft.com/en-in/services/
```

开始使用 Microsoft Azure 之前，你需要先注册一个账户。由于本章主要介绍如何在 Azure 上实现我们的应用程序，所以不会涉及如何创建账户的内容。如果你确实需要帮助，

可以阅读下面的链接给出的文章：

`https://medium.com/appliedcode/setup-microsoft-azure-account-cbd635ebf14b`

Azure 提供了一些基于 SaaS 的服务，你可以通过 `https://azuremarketplace.microsoft.com/en-us` 上面的介绍来了解。

Microsoft Azure 基本知识

登录到 Azure 账户后，你将被重定向到 Azure 服务的 Azure 门户（`https://portal.azure.com`）。Azure 将为你提供一个免费账户，并提供 30 天内 200 美元的试用金。Microsoft Azure 也相信即付即用模式，当你用完所有试用金时，你可以切换到付费账户。

以下是 Azure 的一些基本概念，在继续学习后面内容之前需要先了解一下这些概念。

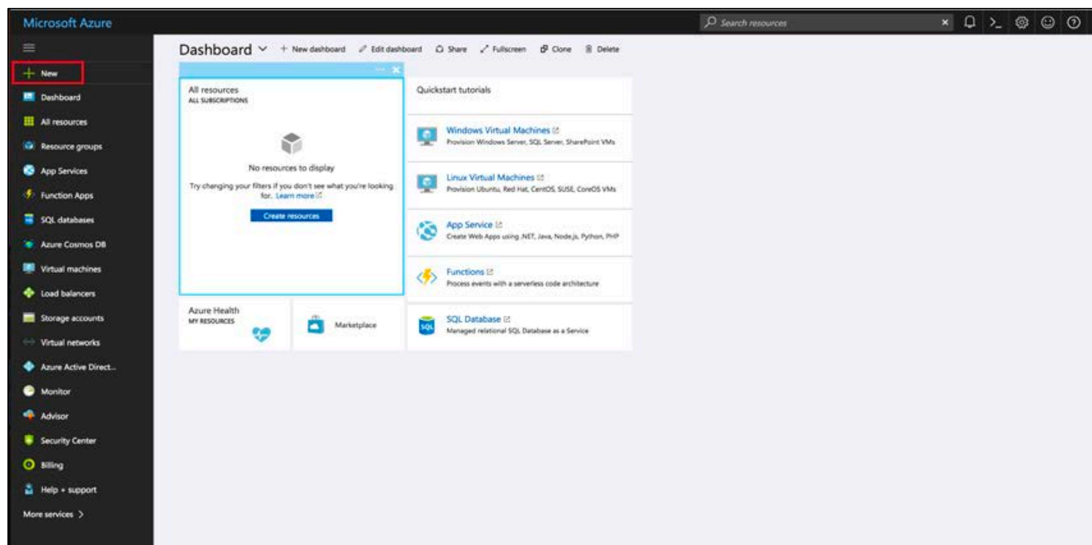
- **Azure 资源管理器**：最初，Azure 基于一种名为 **ASM**（**Azure 服务管理器**）的部署模型。在最新版本的 Azure 中，采用了 **ARM**（**Azure 资源管理器**），它提供了高可用性和更大的灵活性。
- **Azure 可用区**：全球约有 34 个地区分布。
  — Azure 用户列表见 `https://azure.microsoft.com/en-us/regions/`。
  — 特定可用区的所有服务列表见
  `https://azure.microsoft.com/en-us/regions/services/`。
- **Azure 自动化**：Azure 为不同的基于 Windows 的工具提供了许多模板，比如 Azure-PowerShell、Azure-CLI 等。你可以在 `https://github.com/Azure` 上找到这些模板。

Azure 是由微软提供的，我们将主要在 Azure 控制台（UI）上操作，并通过它创建资源。对于喜欢在 Windows 系统上部署应用程序的开发人员或 DevOps 专业人员来说，Azure 环境非常友好，而且它们的应用程序是用.NET 或 VB 编写的。它还支持最新的编程语言，如 Python、ROR 等。

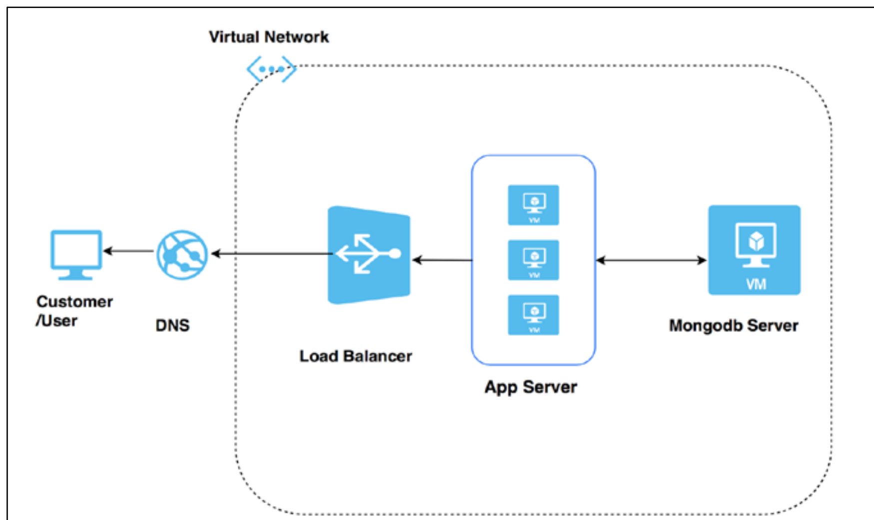
对于喜欢使用 Microsoft 产品（如 Visual Studio）的人来说，Microsoft Azure 是理想的选择。

使用 Azure 构建应用程序的基础架构

在 Azure 门户上，你会在页面上看到如下图所示的默认仪表板。



我们可以在 Azure 上构建应用程序基础架构。我们将按照下图所示的架构在 Azure 上创建生产环境。



在该架构中，我们将使用到如下一系列 Azure 服务。

- **虚拟机**：这与 AWS 中的 EC2 机器类似。我们将会在虚拟机上部署应用程序和 MongoDB 服务。
- **虚拟网络**：虚拟网络与 AWS 中的 VPC 是同义词，为了保持通信网络安全，需要创建虚拟网络。
- **存储**：每个虚拟机都由一个并非我们显式创建的存储账户支持，因为它是与虚拟机一起创建的，用于存储数据。
- **负载均衡器**：与 AWS 中的负载均衡器的使用方式相同，但是在算法上略有差异，因为 Azure 主要遵循基于散列的平衡或源 IP 算法，而 AWS 遵循 Round-Robin 算法或黏滞会话算法。
- **DNS**：如果我们有注册域名，DNS 就会很有用，而且我们需要在 Azure 上管理 DNS。在云平台上，我们称之为区域（Zone）。
- **子网**：我们将在虚拟网络内部创建一个子网来区分资源，根据需要连接互联网。
- **自动扩容**：在该架构图中没有提到这一点，因为这取决于你的应用需求和客户反馈。

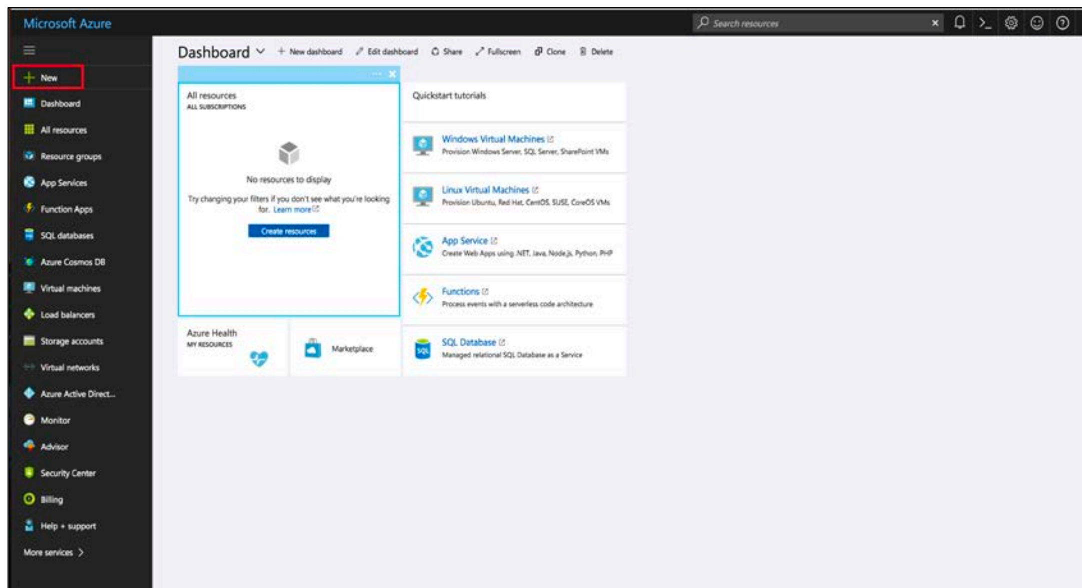
下面，我们开始创建应用程序需要使用的服务器（虚拟机）。

正如前面提到的，Azure 具有非常友好的用户界面，它可以根据你定义的资源在后台创建程序代码，并使用资源管理器把资源提供给你，这使得 DevOps 人员的工作更加轻松。

在 Azure 中创建虚拟机

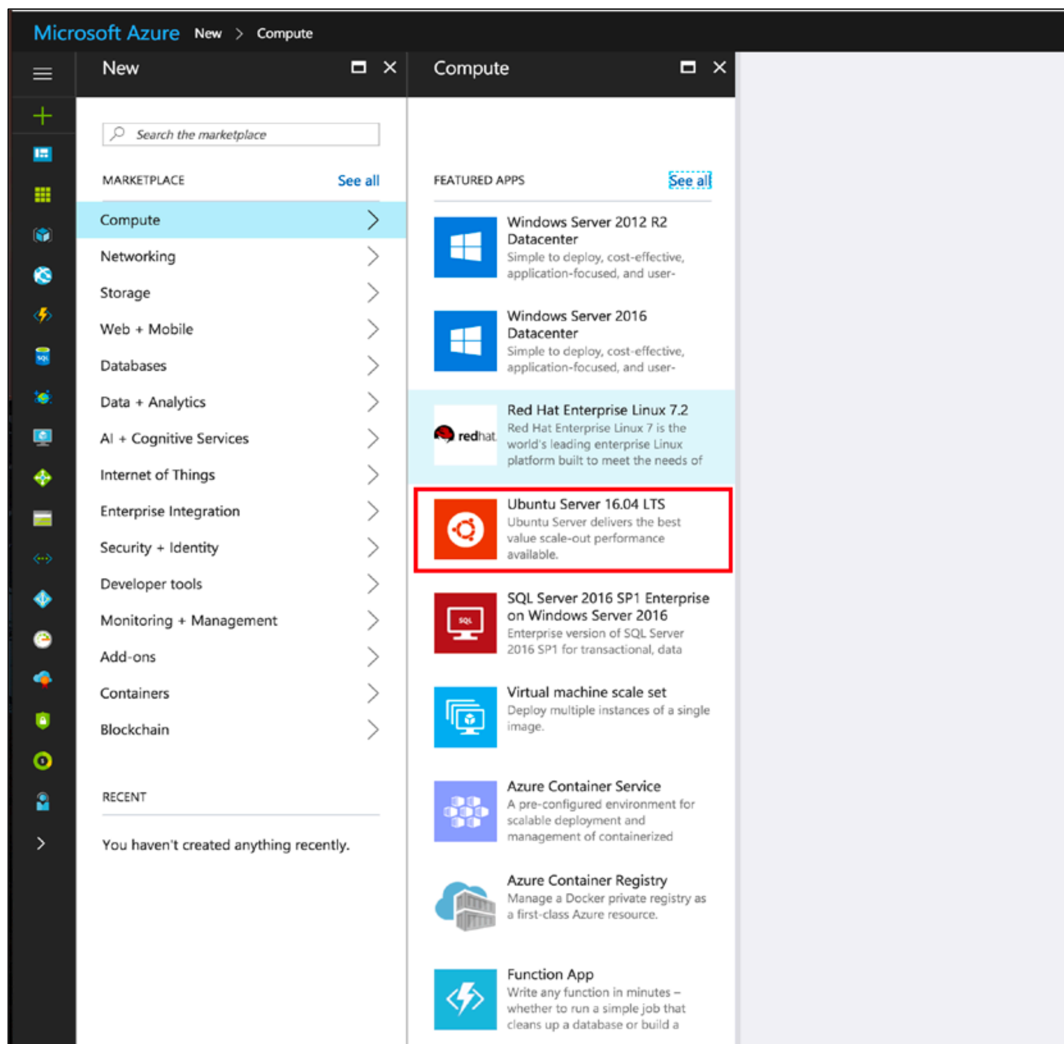
按照下面的步骤在 Microsoft Azure 中创建一个虚拟机。

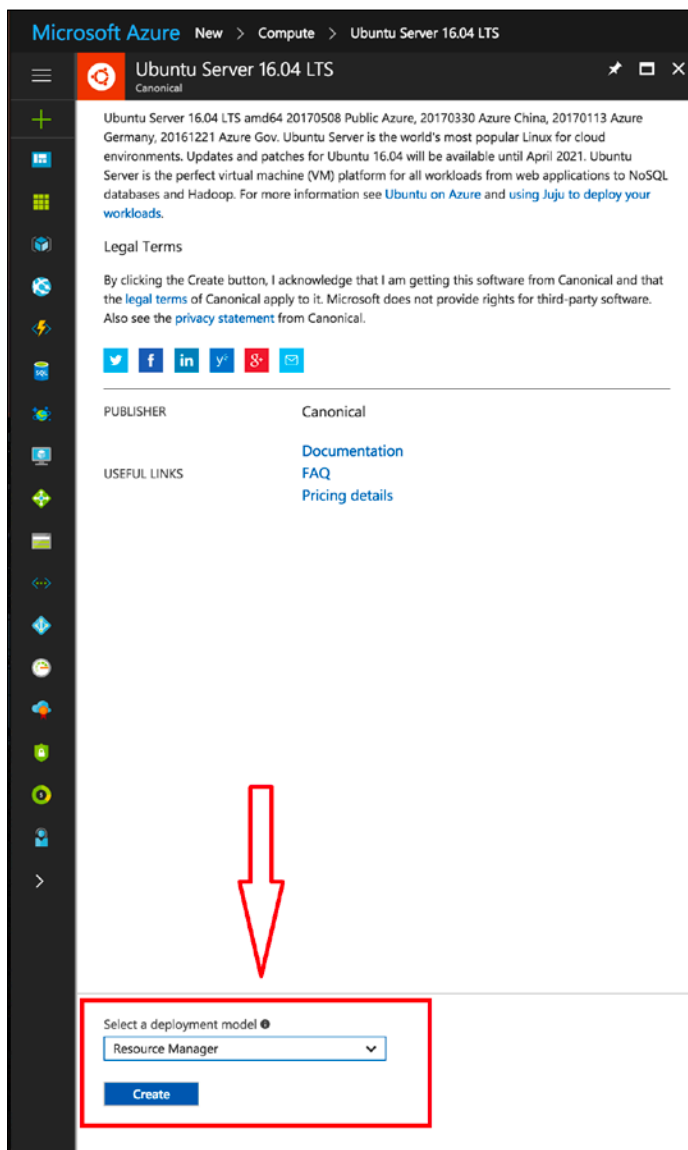
1. 转到 Azure 仪表板，然后在左侧面板中选择 New 以启动 VM Wizard，如下图所示。



2. 接下来需要选择要启动的操作系统。我们将在列表中选择 **Ubuntu Server 16.04 LTS** 服务器（之所以选择此选项，是因为我们的应用程序是在 Ubuntu OS 上开发的）。

在下图所示的屏幕中，需要选择部署模型。有两种可用的部署模型：经典型的（一个标准的 VM）和资源管理型的（高可用性的 VM）。选择 **Resource manager**（资源管理器）模型，如下图所示，然后单击 **Create** 按钮以继续。

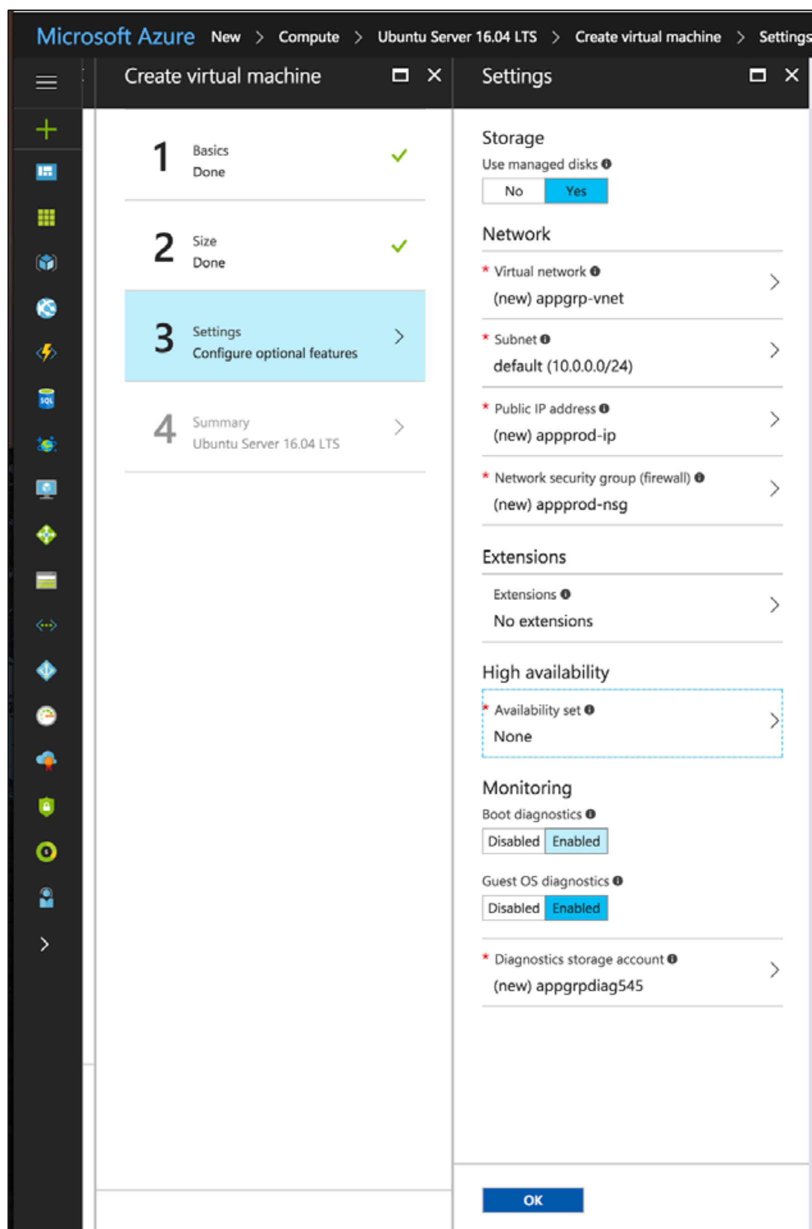



Microsoft Azure New > Compute > Ubuntu Server 16.04 LTS
Ubuntu Server 16.04 LTS
Canonical
Ubuntu Server 16.04 LTS amd64 20170508 Public Azure, 20170330 Azure China, 20170113 Azure Germany, 20161221 Azure Gov. Ubuntu Server is the world's most popular Linux for cloud environments. Updates and patches for Ubuntu 16.04 will be available until April 2021. Ubuntu Server is the perfect virtual machine (VM) platform for all workloads from web applications to NoSQL databases and Hadoop. For more information see Ubuntu on Azure and using Juju to deploy your workloads.
Legal Terms
By clicking the Create button, I acknowledge that I am getting this software from Canonical and that the legal terms of Canonical apply to it. Microsoft does not provide rights for third-party software. Also see the privacy statement from Canonical.
PUBLISHER
Canonical
USEFUL LINKS
Documentation
FAQ
Pricing details
Select a deployment model
Resource Manager
Create

3. 在下一个屏幕上，需要提供 VM 的用户名和认证方法，如下图所示。单击 OK 按钮继续。

4. 接下来，需要根据需求选择 VM 的大小。我们将使用 **DS_V2 标准类型**。选择它，然后单击页面底部的 **Select** 按钮，如下图所示。

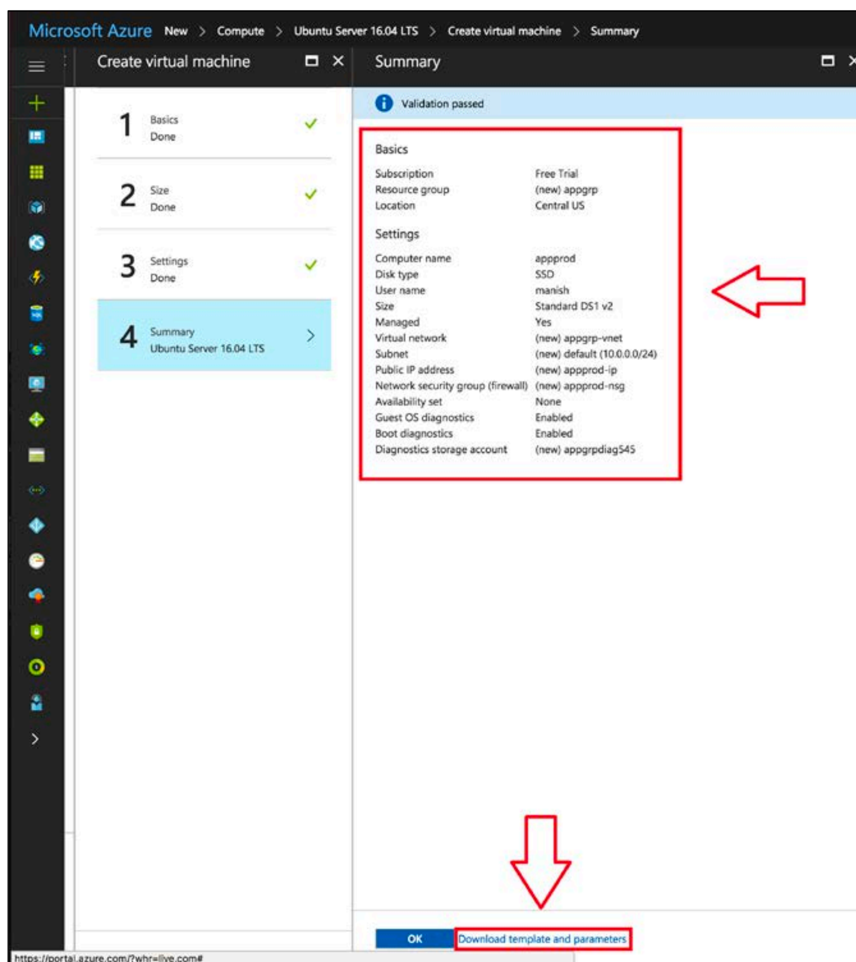
5. 在下一个屏幕中，将定义一些详细信息，如 **Network**（网络）、**Subnet**（子网）、**Public IP address**（公共 IP 地址）、**security group**（安全组）、**Monitoring**（监控）等，如下图所示。



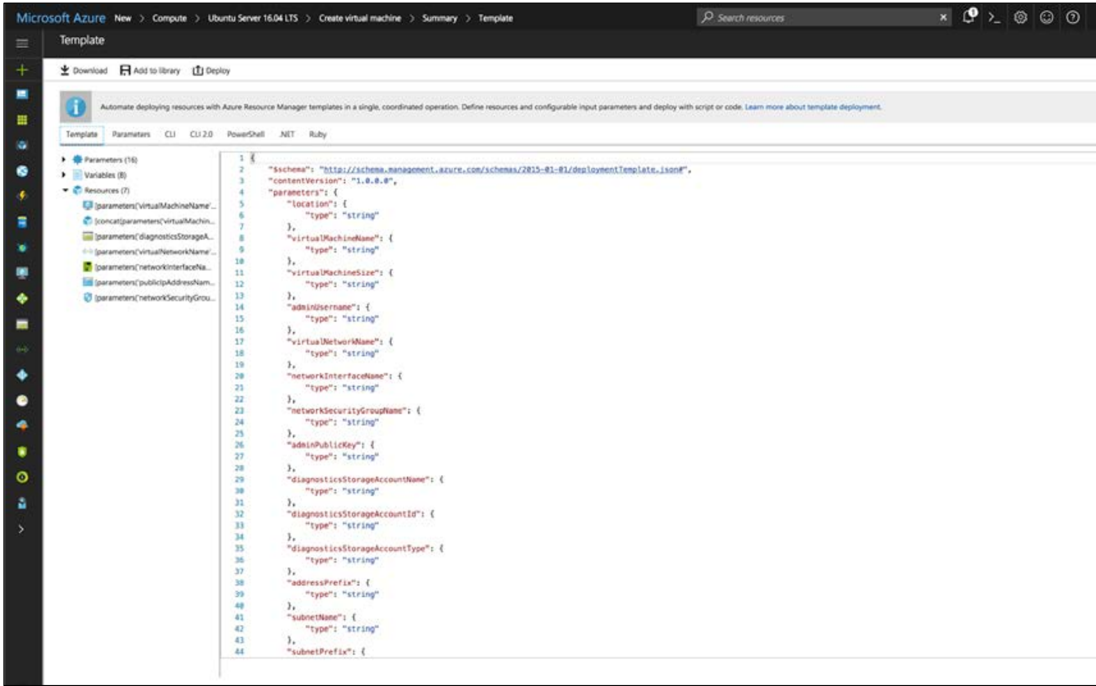
不建议每次创建虚拟机都创建一个虚拟网络，可以在线选择一个虚拟网络。当涉及托管和非托管磁盘时，笔者更喜欢托管的磁盘。这是因为在不受管理的磁盘中，我们选择创建一个存储账户，并且由于我们正在为多个应用程序服务器创建磁盘，因此每个应用程序服务器都有其独立的存储账户。所有的存储账户很可能会落入一个存储单元，这可能导致单点故障。另一方面，Azure 通过单独的存储单元中的每个存储账户来管理磁盘，这使其具有高可用性。

如果你不提供这些详细信息，它会自动设置。

6. 在下一个屏幕中，可查看我们在向导中定义的所有细节，如下图所示。

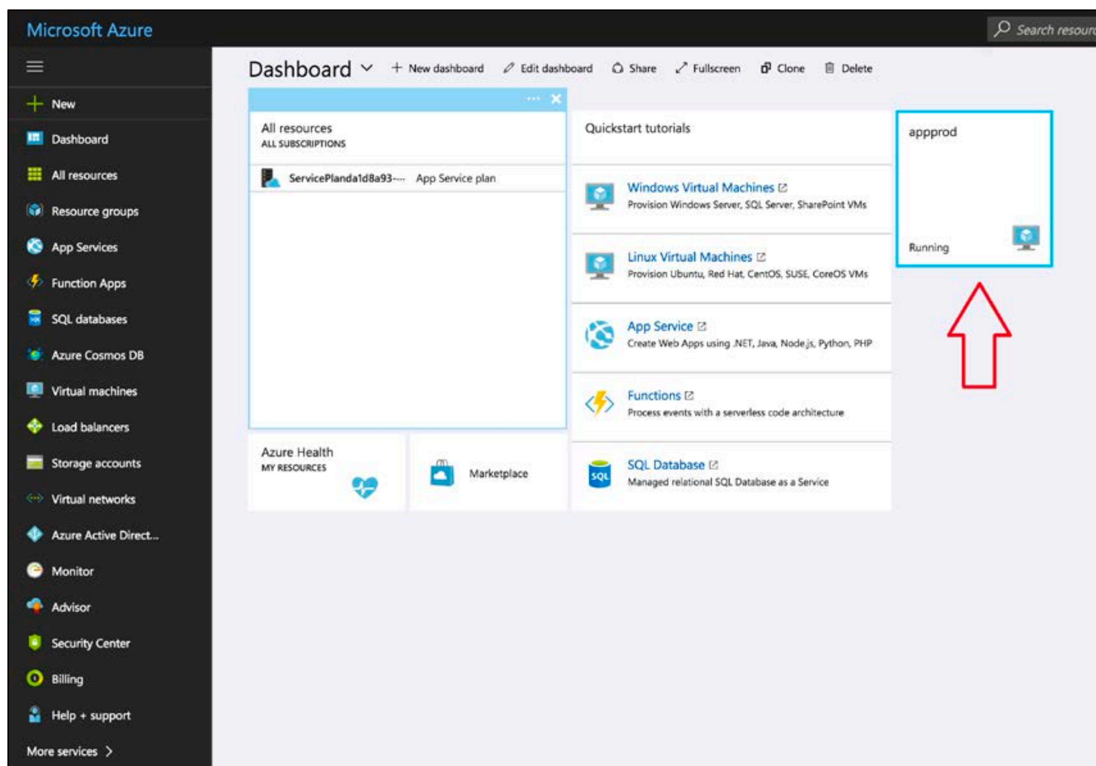


7. 在页面底部，你将找到一个链接，可通过链接以模板的形式或以不同语言的代码形式下载完整的配置。下图所示的屏幕中显示了对我们提供的配置生成的代码。



8. 单击 OK 按钮开始虚拟机的部署。

等一段时间后，从仪表板中应该可以看到有一台虚拟机正在运行，如下图所示。

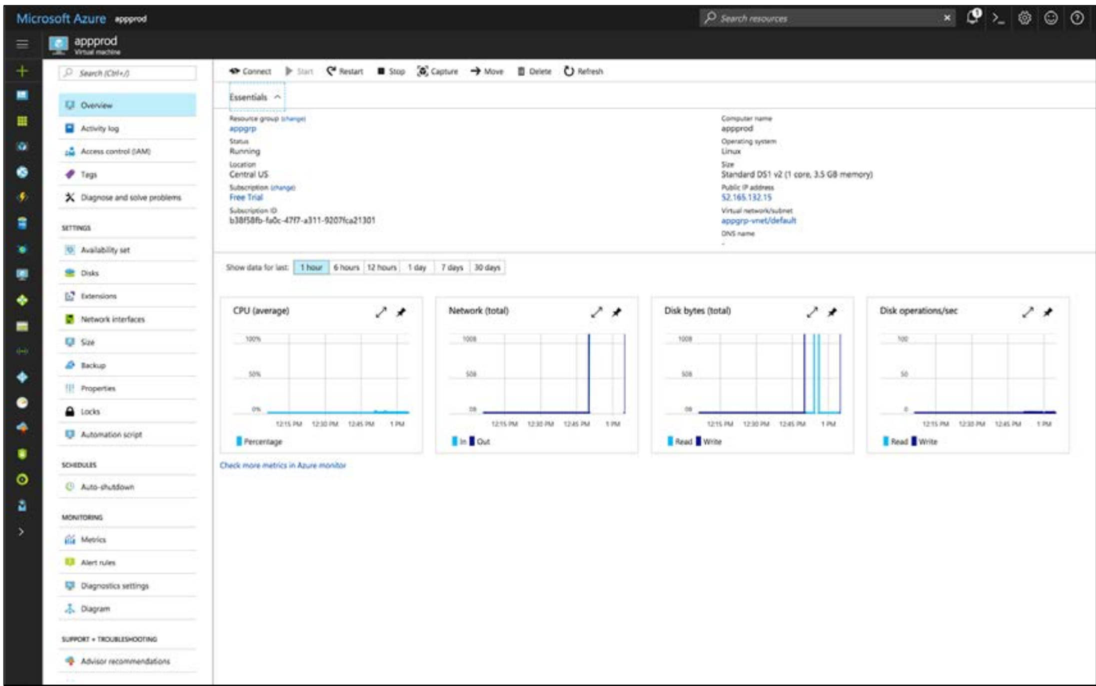


现在你可以访问虚拟机，你需要下载应用程序并像在本地计算机上那样部署它。

同样，可以为应用程序创建多个 VM 实例，作为应用程序服务器。

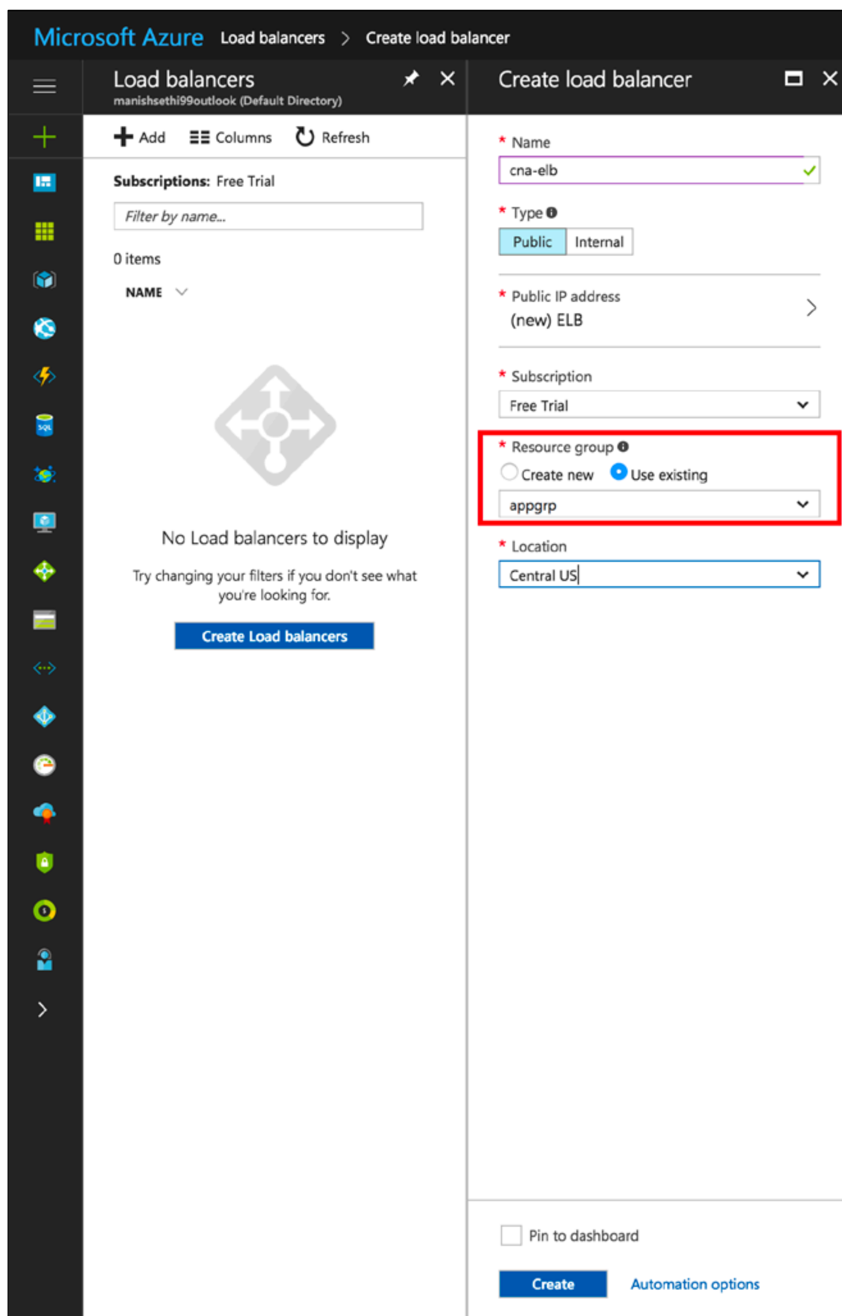
另外，可以在其上 **Greate Load balancers**（创建一个安装了 MongoDB 服务器）的虚拟机。步骤与我们在第 4 章中介绍的步骤类似。

通过单击仪表板上的 VM（即 **appprod**）图标，可以看到 VM 的性能，如下图所示。



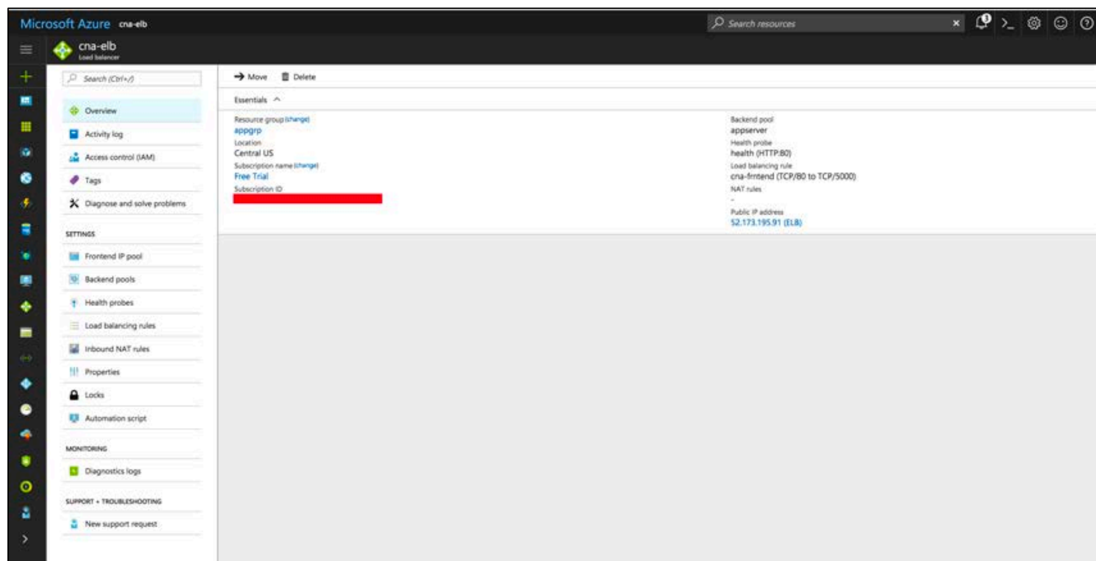
接下来，需要将之前创建的应用程序服务器添加到负载均衡器上。所以，我们需要按照以下步骤创建一个负载均衡器。

- 转到 `https://portal.azure.com/?whr=live.com#blade/HubsExtension/Resources/resourceType/Microsoft.Network%2FLoadBalancers`，单击屏幕中间的 Create Load Balancers（创建负载均衡器）按钮，如下图所示。
- 在下一个屏幕中，需要指定 LB 名称，并提供 LB 用途类型。可以在与应用程序服务器相同的组中启动 ELB，如下图所示。

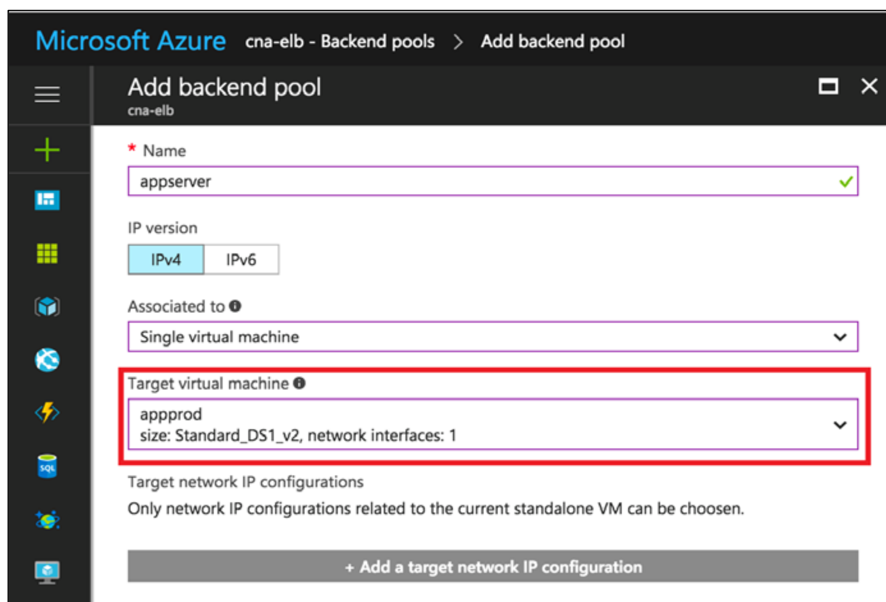


单击 **Create** 按钮开始创建 LB。

9. 负载均衡器准备就绪后，我们应该可以看到下图所示的屏幕，其中显示了负载均衡器的详细信息。



10. 接下来，需要添加一个应用程序服务器后台池，如下图所示。



11. 接下来需要添加健康状况探针，用来指示应用程序的健康状况，如下图所示。

Microsoft Azure cna-elb - Health probes > Add health probe

Add health probe
cna-elb

* Name
health

Protocol
HTTP TCP

* Port
80

* Interval
5
seconds

* Unhealthy threshold
2
consecutive failures

接下来，我们为应用程序添加前端池，如下图所示。

Microsoft Azure　cna-elb - Load balancing rules > Add load balancing rule

Add load balancing rule
cna-elb

* Name
cna-frntend

* Frontend IP address
52.173.195.91 (LoadBalancerFrontEnd)

Protocol
TCP　UDP

* Port
80

* Backend port
5000

Backend pool
appserver

Health probe
health (HTTP:80)

Session persistence
None

Idle timeout (minutes)
4

Floating IP (direct server return)
Disabled　Enabled

现在我们已经为应用程序设置了**负载均衡器**。

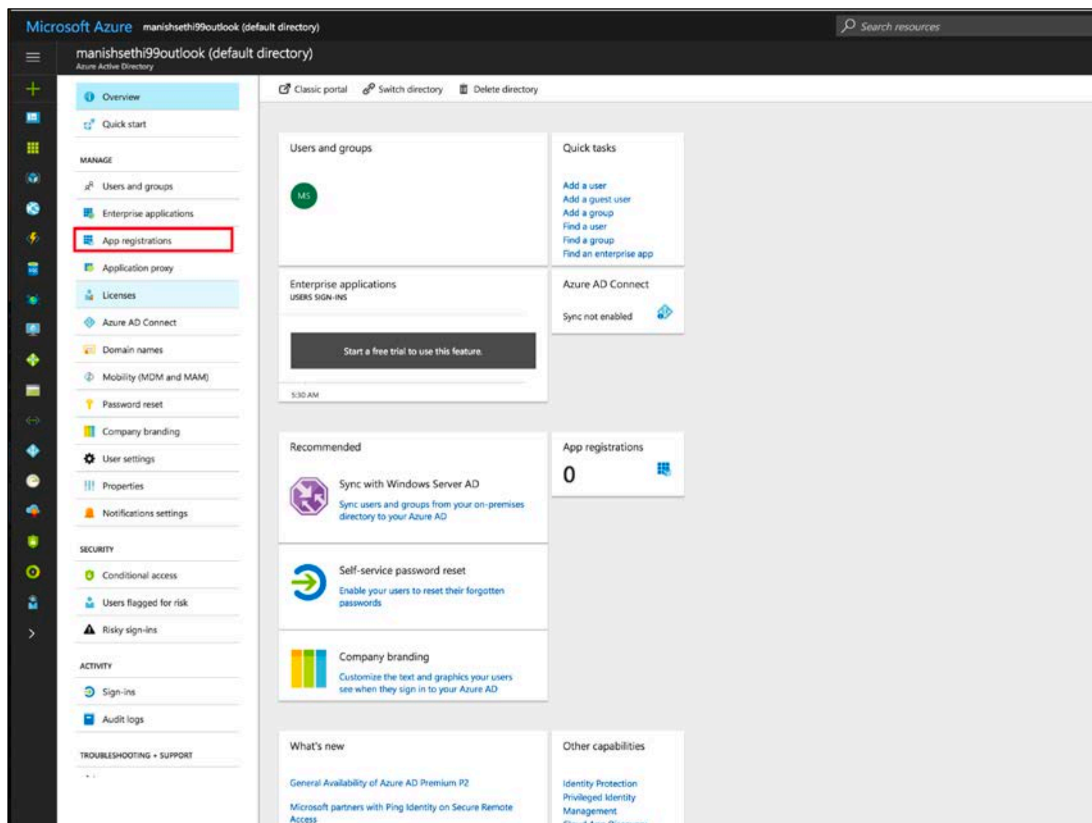


你可以在此链接中阅读有关 Azure 中的负载均衡器的更多信息：`https://docs.microsoft.com/en-us/azure/load-balancer/load-balancer-overview`。

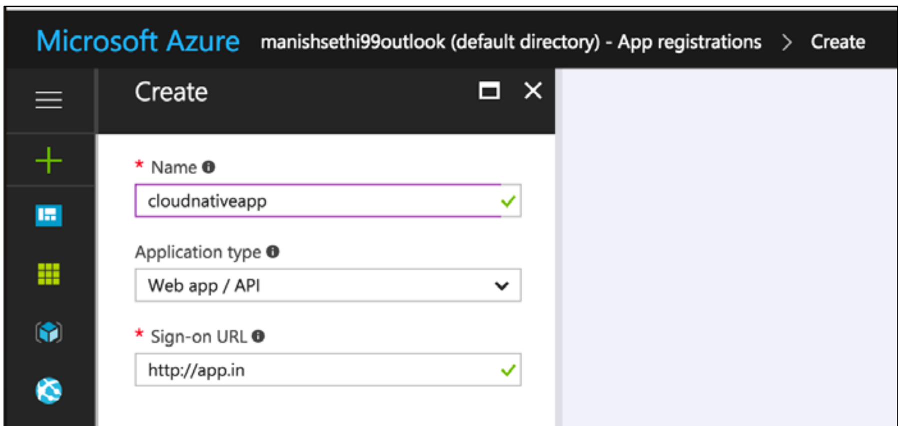
至此，我们已经根据架构图创建了基础架构。下面可以配置 Jenkins，以在 Azure 中的基础架构上部署应用程序。

在 Azure 中使用 Jenkins CI/CD 流水线

首先，需要导航到活动目录服务，如下图所示。

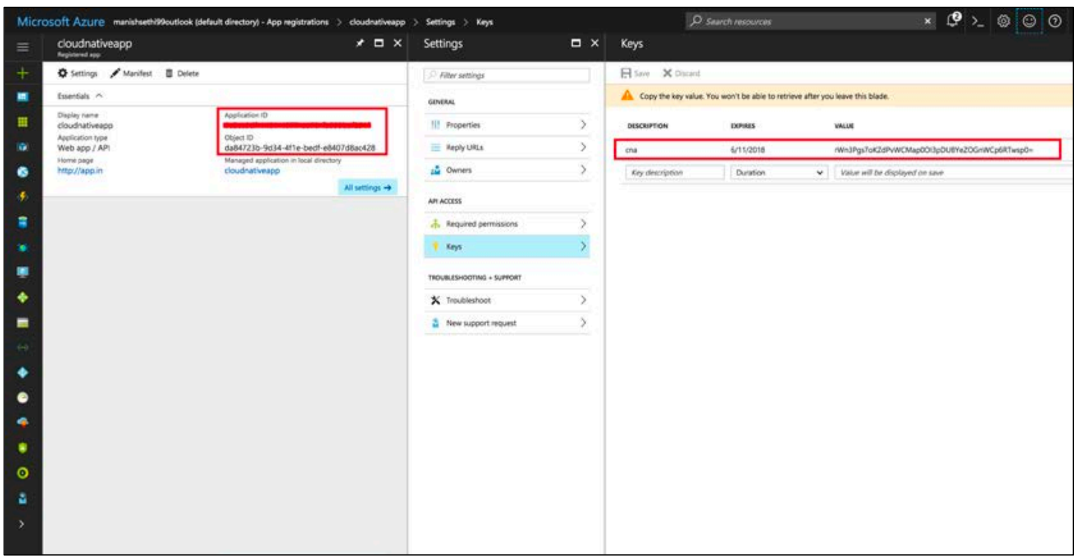


现在需要注册我们的应用程序，选择左侧面板中的 **App registrations（应用程序注册）**链接。你将看到如下图所示的画面，在这里需要提供应用程序的详细信息。



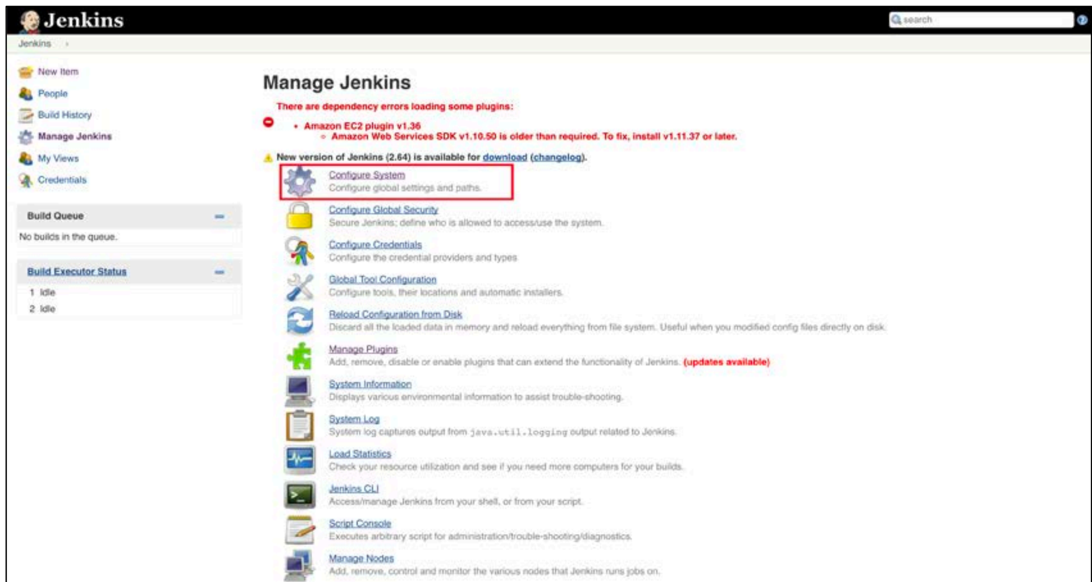
1. 之后，能够生成访问 Jenkins job 所需的密钥。

2. 下图所示的屏幕中包含密钥的详细信息，你还可以在此页面上找到其他详细信息，例如对象 ID 和应用程序 ID 等。



现在我们有了配置 Jenkins job 的所有信息。导航到 Jenkins 控制台，在 **Manage Jenkins（管理 Jenkins）**部分**管理插件**，然后安装插件、**Azure VM 代理**。

插件安装完成后，进入 Manage Jenkins，单击 **Configure System**，如下图所示。



在下一个屏幕中，滚动到 Cloud 部分的底部，单击 **Add cloud** 按钮，然后选择新的 **Microsoft Azure VM Agents** 选项。之后，在同一页面上会出现一个新的部分，如下图所示。



你可以阅读链接（https://wiki.jenkins.io/display/JENKINS/Azure+VM+Agents+plugin）上的文档了解有关微软 Azure VM Agents 插件的更多信息。

在最后一个屏幕中，需要添加之前生成的 Azure 凭证。单击 **Add** 按钮，添加诸如 **Subscription ID**（**订阅 ID**）等的值，如下图所示。

Jenkins Credentials Provider: Jenkins

Add Credentials

Domain Global credentials (unrestricted)

Kind Azure Publisher Settings

Scope Global (Jenkins, nodes, items, all child items, etc)

Publisher Settings Choose file No file chosen Get it from microsoft.com

Subscription ID

Subscription Name

Service Management URL https://management.core.windows.net

Service Management Certificate

ID

Description

Add Cancel

在此页面中，还需要提供虚拟机的详细信息（如模板、VM 类型等），如下图所示。

Add Azure Virtual Machine Template

General Configuration

Name defaulttemplate

Description

Agent Workspace

Labels msubuntu

Region South India

Virtual Machine Size Standard_DS1

Storage Account Type Standard_LRS

Storage Account Name

(Leave blank to create a new storage account)

Retention Time (in minutes) 60

Shutdown Only (Do Not Delete) After Retention Time

Usage Use this node as much as possible

Image Configuration

Custom User Image

Image Reference

Image Publisher Canonical

Image Offer UbuntuServer

Image Sku 16.04-LTS

Image Version latest

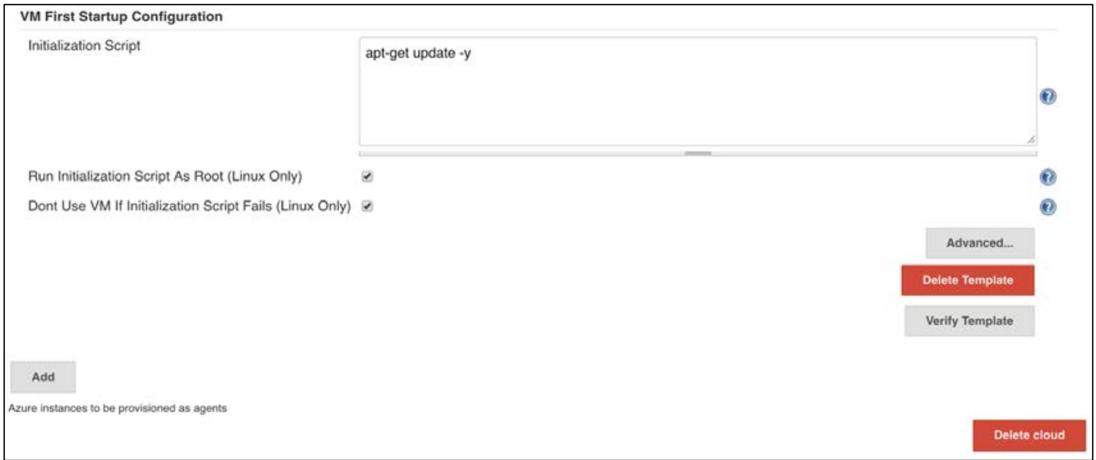
OS Type Linux

Launch Method SSH

Admin Credentials admin/****** Add

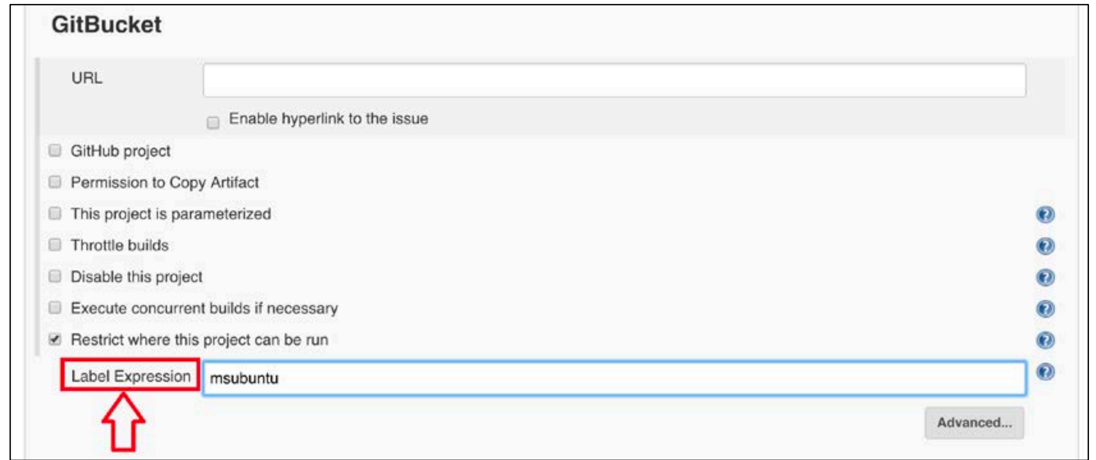
在上面的图中，**Labels** 是最重要的属性，我们将在 Jenkins job 中使用它来标识组。

现在需要提供你想要执行的操作，也就是说，如果你要部署应用程序，则可以提供下载代码和运行应用程序的命令，如下图所示。



单击 **Save** 按钮应用设置。

现在，在 Jenkins 中创建一个新的 job。此外，在通常提供存储库详细信息的 **GitBucket** 部分中，可以找到一个新的复选框，**Restrict where this project can be run（指出运行此项目的位置）**，并要求你提供 **Label Expression（标签表达式）**的名称。在我们的例子中是 **`msubuntu`**，如下图所示。



现在准备运行我们的 Jenkins job，将应用程序部署到 VM（即应用程序服务器）上。

最后，我们可以在 Azure 平台上部署我们的应用程序了。

本章小结

在本章中，我们介绍了微软提供的 Azure 平台，并在其上部署了云原生应用程序。还介绍了在 Azure 平台上构建相同基础架构的不同方法，以及如何在 Azure 平台上集成 Jenkins CI/CD。在下一章即最后一章中，我们将介绍几款用于管理和解决应用程序相关问题非常有用的工具，使用这些工具能够更快地解决问题，使我们的应用程序保持零停机时间。请继续关注下一章的内容！

13
监控云应用

在前面章节中，我们讨论了云原生应用程序的开发，并介绍了如何将其部署到云平台中以满足高可用的需求。到这里，我们的工作还没有完成。应用程序管理和基础设施是一个单独的部分或流程，其监控基础设施和应用程序的性能，使用工具来最小化停机时间甚至达到零停机时间。在本章中，我们将讨论这些监控工具。

本章将涵盖以下主题：

- AWS 服务，如 CloudWatch、Config 等
- Azure 服务，如 Application Insight、Log Analytics 等
- 用于日志分析的 ELK 技术栈
- 开源监控工具，如 Prometheus

云平台上的监控

在前面章节中我们讨论了如何开发应用程序并将其部署到不同的云平台上，以实现我们的业务逻辑。然而，即使在应用程序开发完了之后，也需要有专人利用各种工具来管理这些部署到公有云或者在内部部署的应用程序。

这一节我们将主要讨论由公有云提供商提供的工具或服务，通过这些工具或服务来管理基础设施并提供应用程序洞察力。

在继续讨论这些工具之前，在为任何应用程序确定基础架构时，需要考虑以下几点：

- 定期对特定的请求进行负载测试是一种很好的做法。这将有助于判断应用程序的初始资源需求。这里推荐两个工具：Locust（`http://locust.io/`）和 JMeter（`https://jmeter.apache.org/`）。
- 建议使用最小的配置来分配资源，并使用自动扩容相关的工具，以根据应用程序使用情况管理资源。
- 在资源分配方面应该尽量少手动干扰。

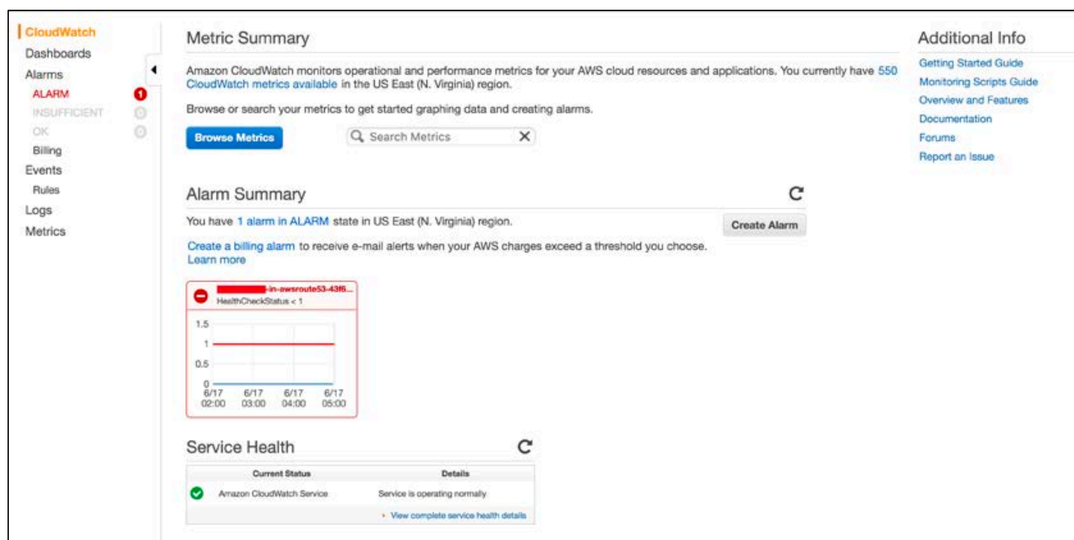
考虑好这几点。因为需要确保有一个监控机制来跟踪资源分配和应用程序性能。下面我们讨论云平台提供的服务。

基于 AWS 的服务

以下是 **AWS**（**Amazon Web Services**）提供的服务和它们在应用程序和基础架构监控方面的使用情况。

CloudWatch

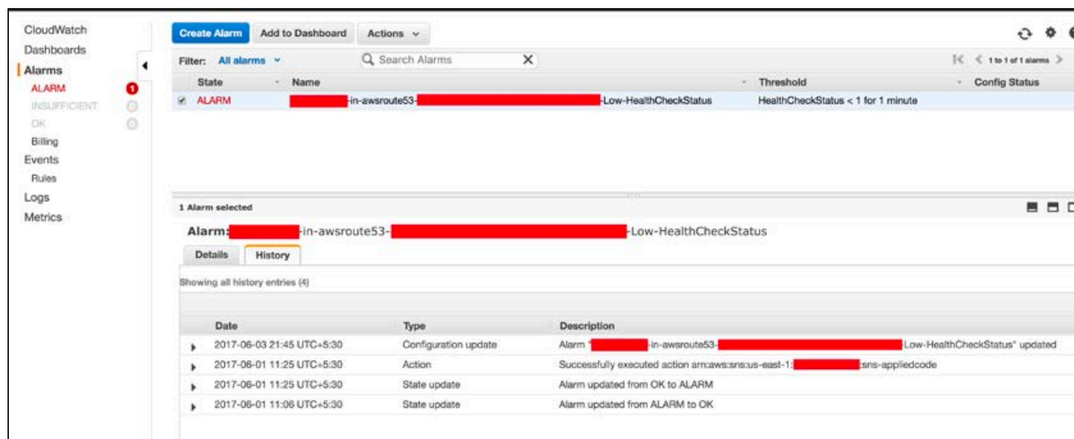
此 AWS 服务会跟踪你的 AWS 资源使用情况，并根据定义的报警配置向你发送通知。可以使用它来跟踪 AWS 账单、Route 53、ELB 等资源。下图显示了一个触发的警报。



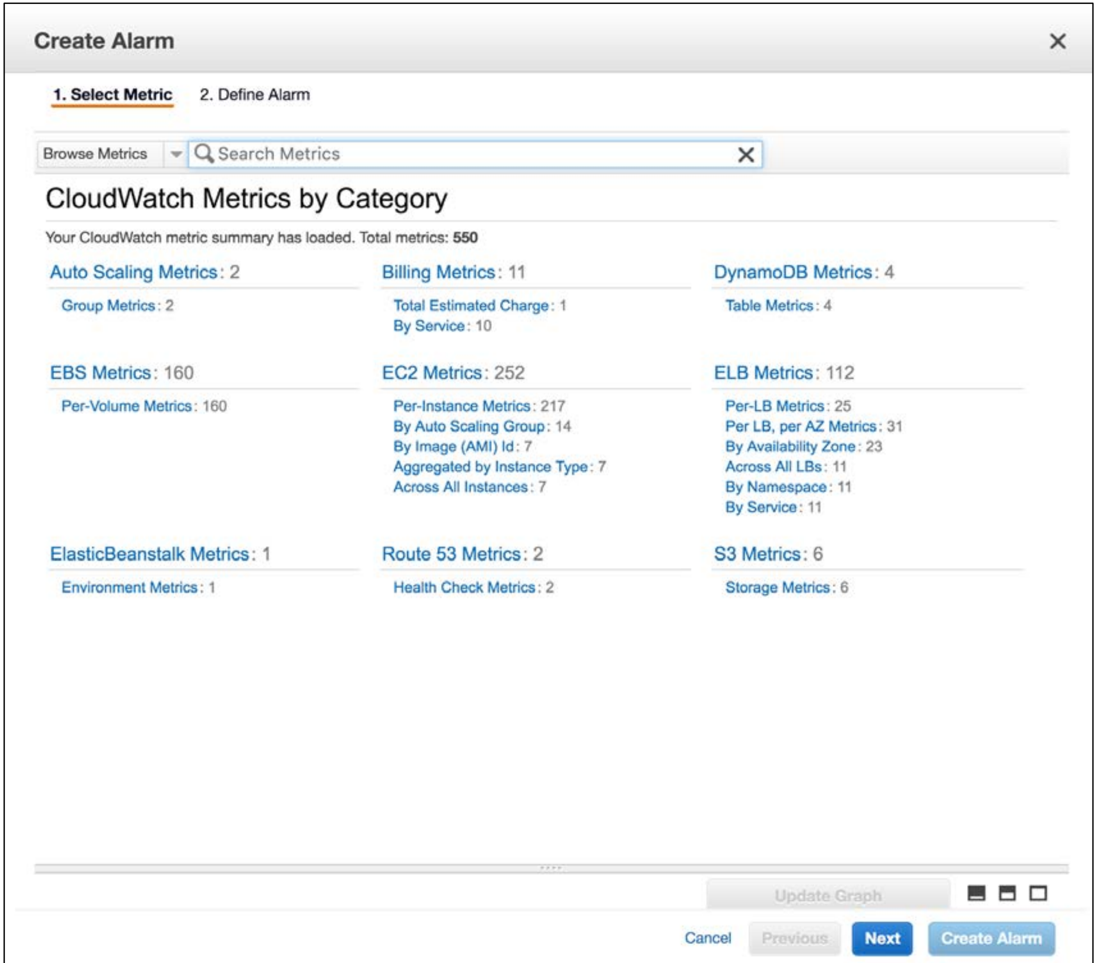
最初，我们需要在下面的链接中设置 CloudWatch 警报。

```
https://console.aws.amazon.com/cloudwatch/home?region=us-east
-1#alarm:alarmFilter=ANY
```

你应该会看到下图所示的屏幕，单击 **Create Alarm** 按钮才能根据指标创建自己的警报。



单击 **Create Alarm** 按钮，你将看到一个弹出式向导，询问你需要监控的指标，如下图所示。



该图列出了所有可用的度量标准，CloudWatch 可以对这些度量标准进行监控并设置警报。

在下图所示的屏幕中，检查 EC2 指标。根据你的需求选择需要监控的指标，例如选择 **NetworkIn** 指标，然后单击 **Next** 按钮。

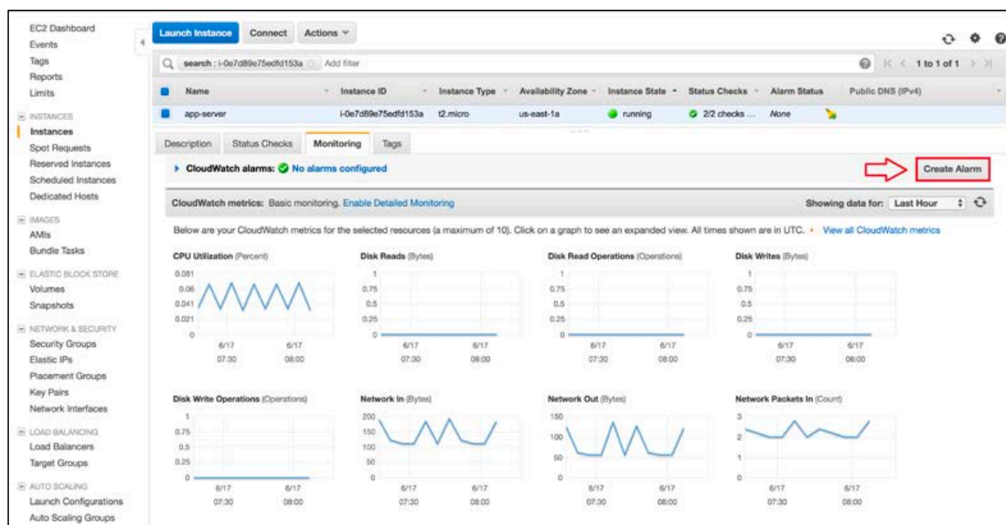
在下一个屏幕上，需要提供警报**名称**和**说明**，同时可以看到**警报预览**。此外，需要提供触发警报的条件信息。

另外，还需要设置通知服务，将通知作为电子邮件发送，如下图所示。

现在，只要 NetworkIn 指标达到其阈值，就会通过电子邮件发送通知。

同样，我们可以设置不同的度量来监视资源利用率。

另一种创建警报的方法是在资源监控部分单击 **Create Alarm** 按钮，如下图所示。



我们可以阅读 AWS 文档（`https://aws.amazon.com/documentation/cloudwatch/`）获取更多信息。

CloudTrail

这是 AWS 中最重要的云服务之一，默认情况下，它会跟踪你的 AWS 账户上的所有活动，无论该活动是通过控制台还是编程访问。在这个服务中不需要配置任何东西。如果你的账户受到攻击，或者你需要检查资源等，才需要配置。

下图显示了与该账户相关的活动。

API activity history
Trails

API activity history

The following list includes the last 7 days of API activity for supported services. The list only includes API activity for create, modify, and delete API calls. For read-only API activity, go to your Amazon S3 bucket or CloudWatch Logs.

You can filter the list using the available attributes, and you can choose an event to see more detail about the event. Learn more.

Filter: Select attribute Enter lookup value Time range: Select time range

Event time	User name	Event name	Resource type	Resource name
2017-06-17, 11:03:21 AM	root	PutMetricAlarm		
2017-06-17, 10:28:48 AM	root	ConsoleLogin		
2017-06-17, 09:39:48 AM	root	ConsoleLogin		
2017-06-15, 03:05:21 PM	root	ConsoleLogin		
2017-06-12, 09:16:30 AM	root	ConsoleLogin		
2017-06-11, 06:20:52 PM	root	StopInstances	EC2 Instance	i-07e66411
2017-06-11, 06:20:40 PM	root	ConsoleLogin		
2017-06-10, 08:58:46 PM	root	StartInstances	EC2 Instance	i-07e66411
2017-06-10, 08:58:40 PM	root	ModifyInstanceAttribute	EC2 Instance	i-07e66411
2017-06-10, 08:58:26 PM	root	StartInstances	EC2 Instance	i-07e66411

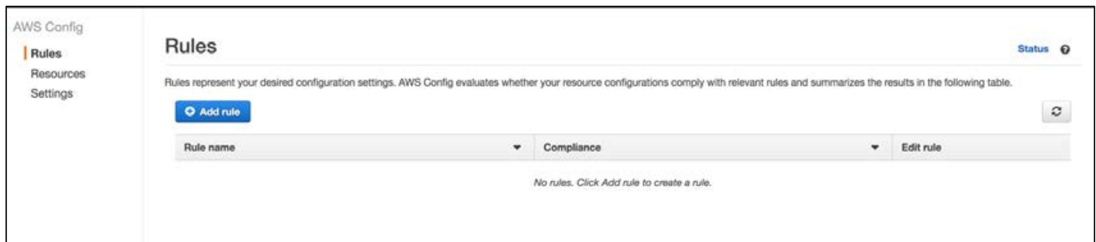
有关更多信息，请阅读 AWS 文档（`https://aws.amazon.com/documentation/cloudtrail/`）。

AWS Config service

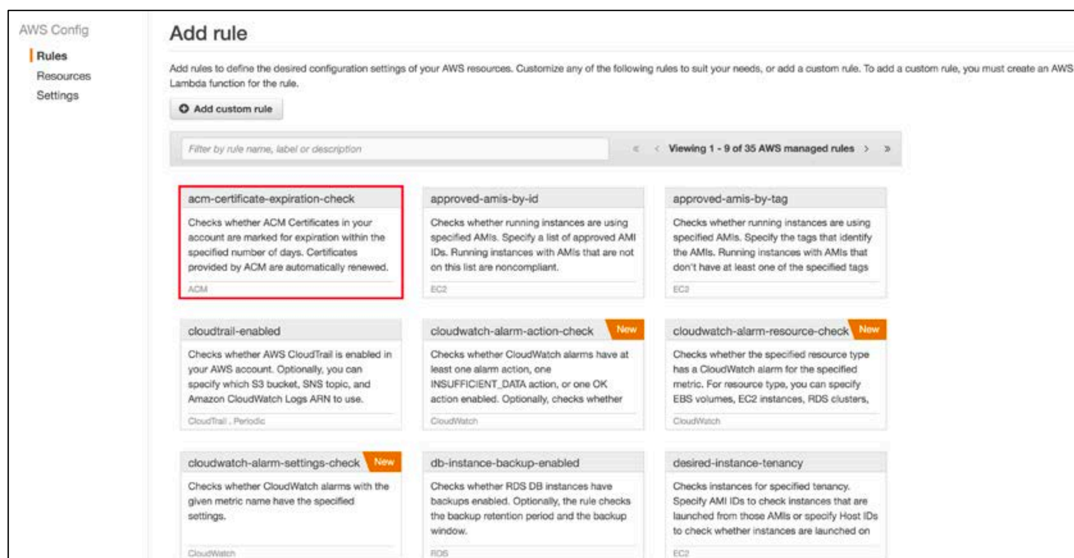
这是另一个 AWS 服务，可以根据定义的模板规则检查 AWS 资源的配置。

> 该服务需要创建用于访问 AWS 资源的服务角色。

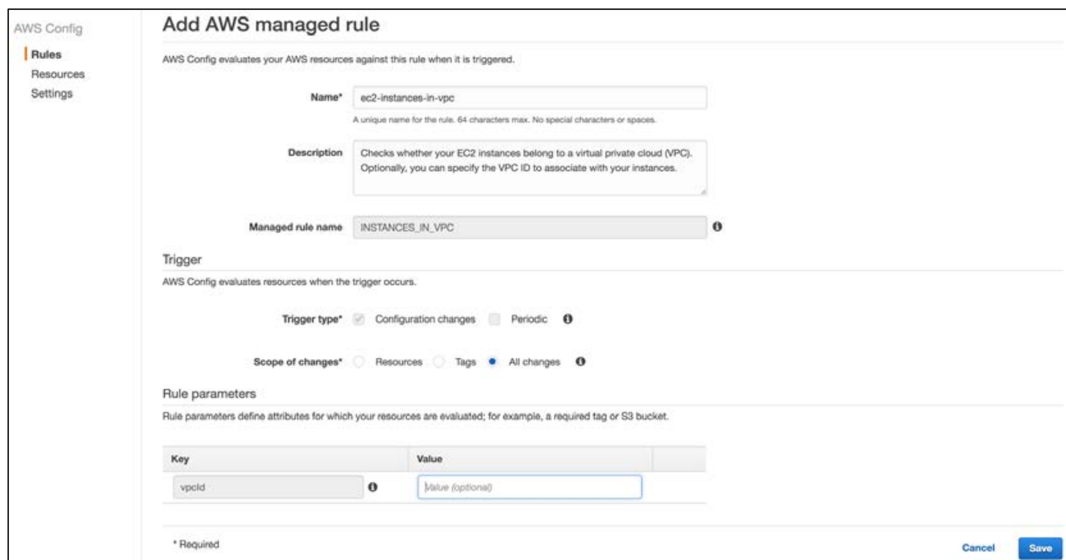
在这个服务中，我们只需要根据提供的模板来设置规则。我们创建的资源作为应用程序部署的一部分，使用 AWS 或客户提供的模板来检查这些资源。为了向服务配置中添加新的规则，请转到 `https://console.aws.amazon.com/config/home?region=us-east-1#/rules/view`，如下图所示。



在该屏幕中，我们需要添加一个新规则，该规则将用于评估所有资源或你指定的资源。单击 **Add rule（添加规则）**按钮添加新规则，如下图所示。

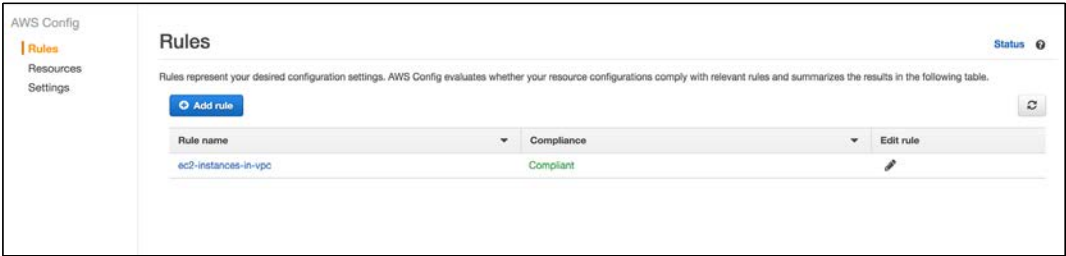


在该屏幕中，基于要跟踪的资源，选择打开资源监控配置的规则。

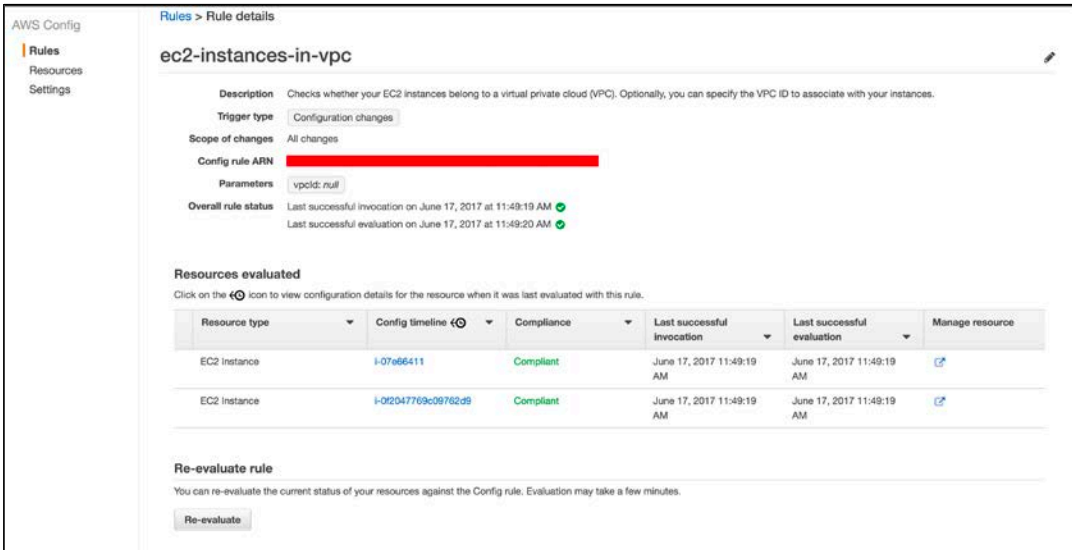


该屏幕针对的是 AWS **ec2-instance-in-vpc** 模板配置，其可以帮助你验证 EC2 是否使用了正确的配置，且已经在 VPC 中。在这里，你可以指定需要评估哪个 VPC。

单击 **Save** 按钮添加一个新的规则。评估完毕后，我们将看到下图所示的屏幕。



资源报告如下图所示。



你可以通过 AWS 文档（`https://aws.amazon.com/documentation/config/`）了解更多信息。

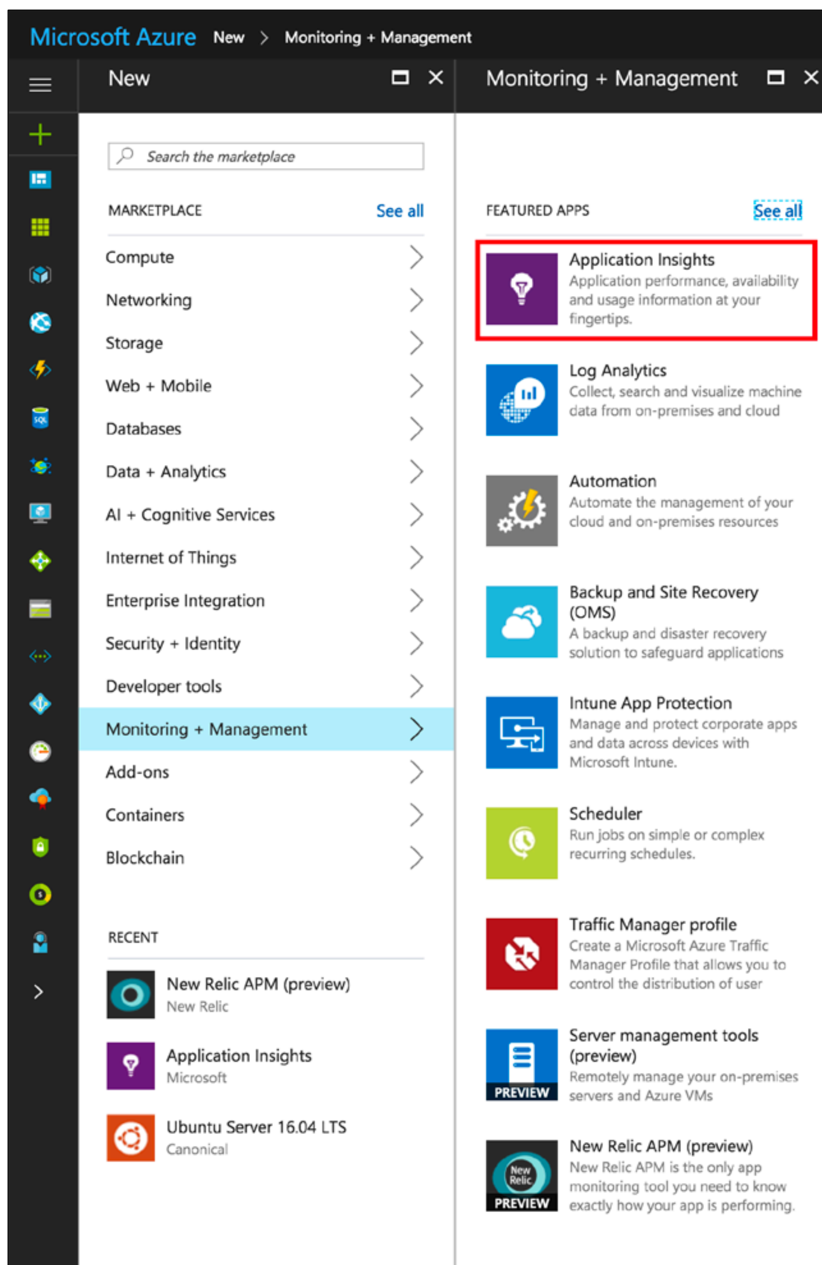
Microsoft Azure 服务

以下是 Microsoft Azure 提供的服务，它可帮助你管理应用程序的性能。

Application Insights

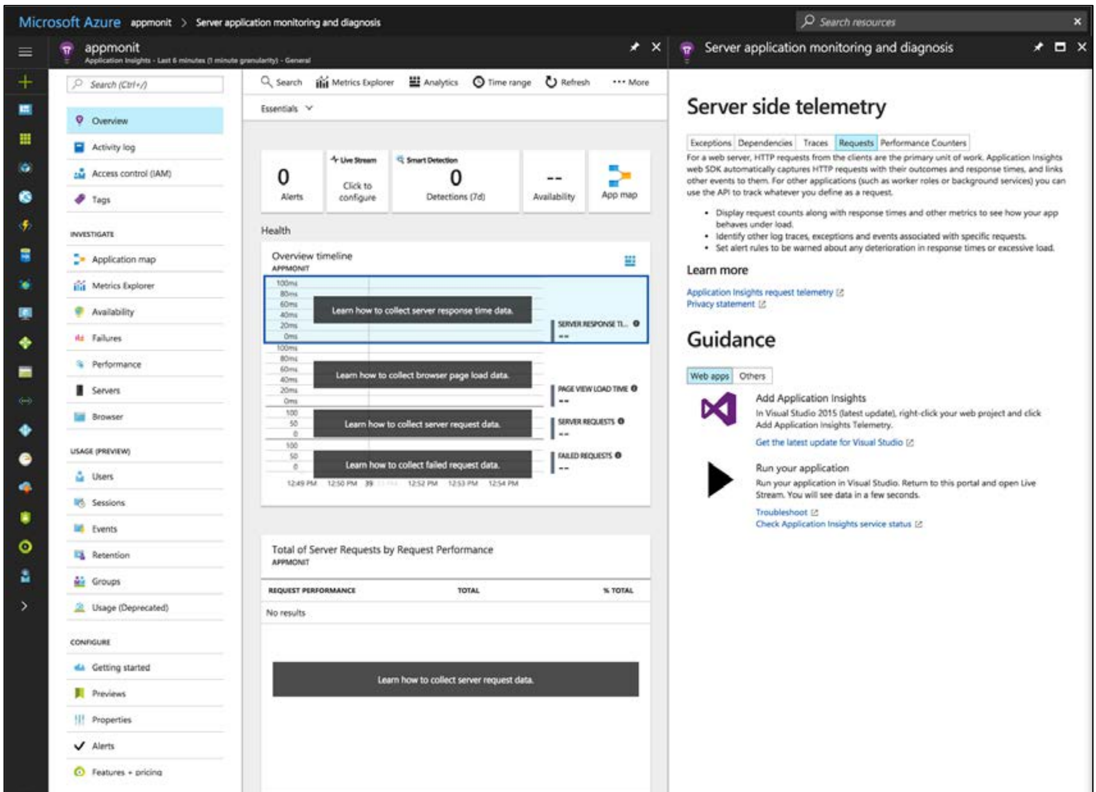
Azure 提供的此服务可帮助你管理应用程序的性能，这对于 Web 开发人员非常有用，可以帮助你检测、分类和诊断应用程序问题。

要设置 Application Insights，只需要知道你的基础架构所在的应用程序和组的名字。现在，如果你单击左侧窗格中的+符号，应该会看到如下图所示的屏幕。



在这里，可以选择 **Application Insights** 服务，我们需要提供 Application Insight 名称、需要监控的组名以及需要启动的区域。

启动后，你将看到如下图所示的屏幕，这里显示了如何使用 Application Insights 配置资源。



请浏览 `https://docs.microsoft.com/en-us/azure/application-insights/app-insights-profiler` 上的完整参考文档，其中包含有关如何使用资源配置 Application Insights 的完整信息。

现在的问题是 Application Insights 该监控哪些指标。以下是一些监控指标的描述。

- **请求率、响应时间和失败率**：有助于你了解请求类型及响应时间，帮助进行资源管理。
- **Ajax 调用**：跟踪速率、响应时间和网页的失败率。

- **用户和会话详细信息**：跟踪用户和会话信息，例如用户名、登录信息、注销信息等。
- **性能管理**：跟踪 CPU、网络和 RAM 的使用情况。
- **主机诊断**：计算 Azure 的资源。
- **异常**：了解服务器和浏览器的异常情况报告。

你可以为自己的系统配置很多指标。获取更多信息，请查看 `https://docs.microsoft.com/en-us/azure/application-insights/app-insights-metrics-explorer` 上的介绍。

你可以阅读 Azure 文档(`https://docs.microsoft.com/en-us/azure/application-insights/`)了解关于 Application Insights 的详细信息。

到目前为止，我们一直在验证和监控云平台上的应用程序及其基础架构。但是，有个非常重要的问题是，如果出现了应用程序问题我们如何来排除？下面我们要讨论的 ELK 技术栈可以帮助你定位问题，可以是系统级别的问题，也可以是应用程序级别的问题。

ELK 技术栈介绍

ELK 技术栈由 Elasticsearch、Logstash 和 Kibana 组成。这些组件一起工作，来收集系统级日志（即 Syslog、RSYSLOG 等）或应用程序级别日志（即访问日志、错误日志等）等所有类型的日志。

对于 ELK 堆栈的设置可以参考这篇文章，在 ELK 堆栈中使用 Filebeat 配置将日志发送到 Elasticsearch：

```
https://www.digitalocean.com/community/tutorials/how-to-install-elasticsearch-logstash-and-kibana-elk-stack-on-ubuntu-14-04
```

Logstash

Logstash 需要安装在要收集日志的服务器上，日志将被发送到 Elasticsearch 以创建索引。

安装 Logstash 后，建议配置位于/etc/logstash 目录下的 `logstash.conf` 文件，使用 Logstash 日志的文件轮换（即`/var/log/logstash/*.stdout`，`*.err` 或`*.log`）或后缀格式，例如数据格式。以下配置模板供你参考：

```
# see "man logrotate" for details
```

```
 # number of backlogs to keep
 rotate 7
 # create new (empty) log files after rotating old ones
 create
 # Define suffix format
 dateformat -%Y%m%d-%s
 # use date as a suffix of the rotated file
 dateext
# uncomment this if you want your log files compressed
compress
# rotate if bigger that size
size 100M
# rotate logstash logs
/var/log/logstash/*.stdout
/var/log/logstash/*.err
/var/log/logstash/*.log {
        rotate 7
        size 100M
        copytruncate
        compress
        delaycompress
        missingok
        notifempty
}
```

为了将日志发送到 Elasticsearch，配置中需要有三个部分，分别名为 INPUT、OUTPUT 和 FILTER，这有助于创建索引。这些部分可以放在单个文件中或者是单独的文件中。

Logstash 事件处理管道以 INPUT-FILTER-OUTPUT 方式工作，每个部分都有自己的优点和用法。

- **Input（输入）**：事件从日志文件中获取的数据。最普通的输入是文件，使用 `tailf` 读取文件；Syslog，从监听的 Syslog 服务的 514 端口读取数据；beat，从 Filebeat 收集事件。
- **Filter（过滤）**：Logstash 中的这些中间层设备根据定义的过滤器对数据库执行某些操作，并将符合条件的数据过滤出来。其中一些是 GROK（基于定义的模式的结构和解析文本）和克隆（通过添加或删除字段来复制事件）等。

- **Output（输出）：**这是我们将过滤的数据传递给定义的输出的最后阶段。可能有多个输出位置，可以将数据传递给更深层次的索引。Elasticsearch 是常用的输出，它非常可靠，易用，它提供方便的平台来保存数据，并且查询起来更容易；graphite，是一个开源工具，用于以图表的形式存储和显示数据。

以下是 Syslog 的日志配置示例。

- Syslog 的 input 部分配置如下：

```
input {
  file {
  type => "syslog"
 path => [ "/var/log/messages" ]
 }
}
```

- Syslog 的 filter 部分配置如下：

```
filter {
  grok {
   match => { "message" => "%{COMBINEDAPACHELOG}" }
  }
 date {
  match => [ "timestamp" , "dd/MMM/yyyy:HH:mm:ss Z" ]
 }
}
```

- Syslog 的 output 部分配置如下：

```
output {
  elasticsearch {
    protocol => "http"
    host => "es.appliedcode.in"
    port => "443"
    ssl => "true"
    ssl_certificate_verification => "false"
    index => "syslog-%{+YYYY.MM.dd}"
    flush_size => 100
  }
}
```

要发送的日志的配置文件通常保存在`/etc/logstash/confd/`目录下。

如果你为每个部分创建单独的配置文件，则需要遵循文件命名约定，例如，输入文件应该命名为 `10-syslog-input.conf`，过滤文件应该命名为 `20-syslog-filter.conf`。同样，对于输出文件，应该命名为 `30-syslog-output.conf`。

如果你想验证你的配置是否正确，则可以执行以下命令：

```
$ sudo service logstash configtest
```

有关 Logstash 配置的更多信息，请参阅 `https://www.elastic.co/guide/en/logstash/current/config-examples.html` 上的示例。

Elasticsearch

Elasticsearch（`https://www.elastic.co/products/elasticsearch`）是一款日志分析工具，可以帮助你基于具有时间戳配置的大量数据流存储和创建索引，从而解决开发人员使用日志识别和定位问题。Elasticsearch 是一款基于 Lucene 搜索引擎的 NoSQL 数据库。

安装 Elasticsearch 后，可以通过单击以下 URL 来验证版本和集群详细信息：`http://ip-address9200/`

输出如下所示。

```
{
  "name" : "Klaw",
  "cluster_name" : "elasticsearch",
  "cluster_uuid" : "k3uzUoIgStm4Bl5heJ0Bag",
  "version" : {
    "number" : "2.4.5",
    "build_hash" : "c849dd13904f53e63e88efc33b2ceeda0b6a1276",
    "build_timestamp" : "2017-04-24T16:18:17Z",
    "build_snapshot" : false,
    "lucene_version" : "5.5.4"
  },
  "tagline" : "You Know, for Search"
}
```

这证明 Elasticsearch 已经启动并正在运行。现在，如果要查看是否创建了日志，可以使用以下 URL 查询 Elasticsearch：`http://ip-address9200/_search?pretty`。

输出如下所示。

```
{
  "took" : 2,
  "timed_out" : false,
  "_shards" : {
    "total" : 6,
    "successful" : 6,
    "failed" : 0
  },
  "hits" : {
    "total" : 2151,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : ".kibana",
      "_type" : "search",
      "_id" : "Cache-transactions",
      "_score" : 1.0,
      "_source" : {
        "sort" : [ "@timestamp", "desc" ],
        "hits" : 0,
        "description" : "",
        "title" : "Cache transactions",
        "version" : 1,
        "kibanaSavedObjectMeta" : {
          "searchSourceJSON" : "{\"index\":\"packetbeat-*\",\"highlight\":{\"pre_tags\":[\"@kibana-highlighted-
field@\"],\"post_tags\":[\"@/kibana-highlighted-field@\"],\"fields\":{\"*\":{}}},\"filter\":[],\"query\":
{\"query_string\":{\"query\":\"type: redis\",\"analyze_wildcard\":true}}}"
        },
        "columns" : [ "type", "method", "path", "responsetime", "status" ]
      }
    }, {
```

单击 `http://ip-address:9200/_cat/indices?v` 链接查看生成的索引。

输出如下所示。

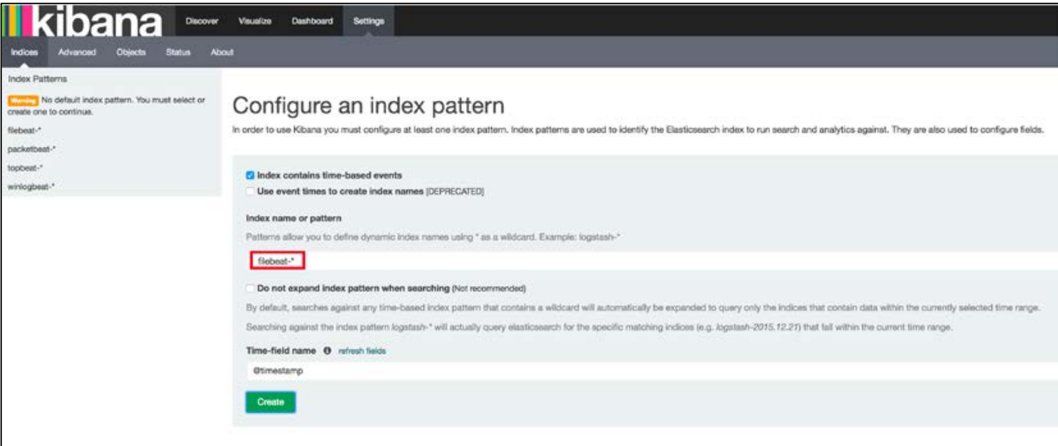
```
health status index               pri rep docs.count docs.deleted store.size pri.store.size
yellow open   .kibana               1   1        103            1     92.8kb         92.8kb
yellow open   filebeat-2017.06.17   5   1       2048            0    576.9kb        576.9kb
```

如果你想了解关于 Elasticsearch 查询、索引等更多信息请参阅这篇文章：`https://www.elastic.co/guide/en/elasticsearch/reference/current/indices.html`。

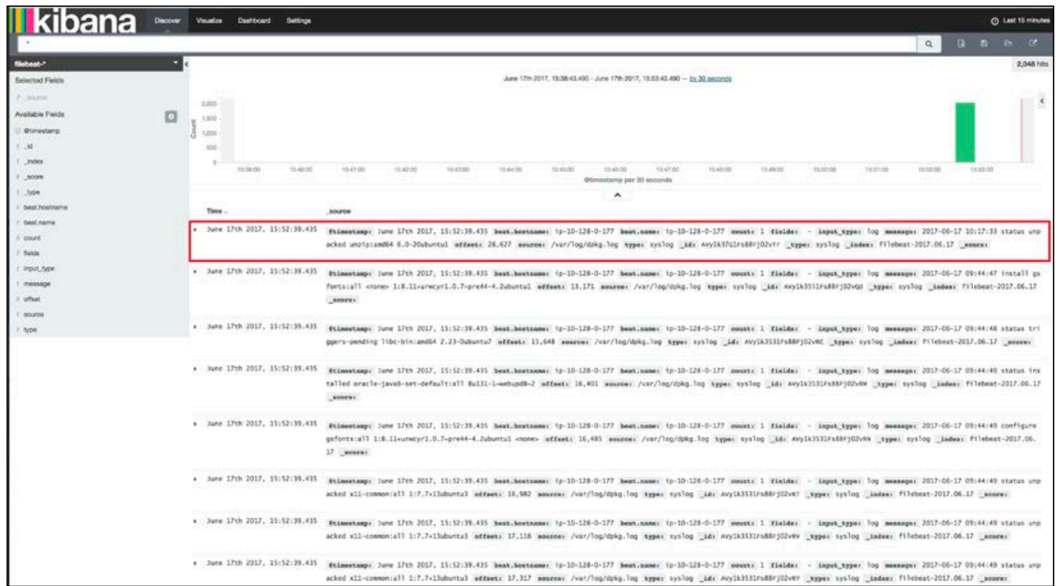
Kibana

Kibana 运行在 Elasticsearch 的上层，其可视化提供对从环境接收到的数据的洞察，并帮助你做出决策。总之，Kibana 是一个 GUI，用于从 Elasticsearch 搜索日志。

安装好 Kibana 之后，访问 `http://ip-address:5601/` 可以看到如下图所示的画面，要求你创建索引并配置 Kibana dashboard。



配置完成后，将看到如下图所示的带有时间戳的日志。



现在，我们需要创建一个仪表板，这个仪表板会给我们提供可视化日志视图，该视图将包括图形、饼图等。

有关创建 Kibana 仪表板的更多信息，请参考 Kibana 文档（`https://www.elastic.co/guide/en/kibana/current/dashboard-getting-started.html`）。

作为 Kibana 的替代方案，你可能会对 Grafana（`https://grafana.com/`）感兴趣，它也是一个分析和监控工具。

问题是 Grafana 和 Kibana 有什么不同呢？下表列出了它们之间的不同点。

Grafana	Kibana
Grafana 仪表板以基于 CPU 或 RAM 系统指标的时间序列图表为主	Kibana 侧重于日志分析
Grafana 的内置 RBA（基于角色的访问）决定用户对仪表板的访问	Kibana 不能控制对仪表板的访问
Grafana 支持除了 Elasticsearch 之外的其他数据源，如 Graphite、InfluxDB 等	Kibana 集成在 ELK 技术栈中

以上我们介绍了 ELK 堆栈，它使我们了解应用程序，并能够帮助我们解决应用程序和服务器的问题。在下一节中，我们将讨论一个名为 **Prometheus** 的开源工具，该工具对于监视不同服务器的活动非常有用。

开源监控工具

这一节，我们将主要讨论第三方工具，并收集服务器的指标以排查应用程序问题。

Prometheus

Prometheus（`https://prometheus.io`）是一个开源监控解决方案，可以持续监控系统活动指标，并在需要采取行动时发出警报。该工具是用 **Golang** 编写的。

该工具类似于 Nagios，并且正在变得流行。它可以收集服务器的指标，但也会根据你的要求提供模板度量指标（如 `http_request_duration_microseconds`），以便你使用 UI 生成图表，从而更好地理解并更有效地监控系统。



Prometheus 默认运行在 `9090` 端口上。

要安装 Prometheus，可按照官方网站（`https://prometheus.io/docs/introduction/getting_started/`）上的说明进行操作。安装完成并启动后打开 `http://ip-address9090/status` 了解状态。如下图所示的屏幕显示了 Prometheus 的版本信息，即版本、修订等。

Prometheus　Alerts　Graph　Status　Help

Runtime Information

Uptime	2017-06-17 11:34:31.742557426 +0000 UTC
Working Directory	/root/prometheus-1.7.1.linux-amd64

Build Information

Version	1.7.1
Revision	3afb3fffa3a29c3de865e1172fb740442e9d0133
Branch	master
BuildUser	root@0aa1b7fc430d
BuildDate	20170612-11:44:05
GoVersion	go1.8.3

Alertmanagers

Endpoint

要想知道配置的目标，请访问 `http://ip-address9090/targets`。输出将如下图所示。

Prometheus　Alerts　Graph　Status　Help

Targets

prometheus

Endpoint	State	Labels	Last Scrape	Error
http://localhost:9090/metrics	UP	instance="localhost:9090"	14.703s ago	

为了生成图形，使用 `http://ip-address9090/graph` 并选择要实现的指标图形。输出应该如下图所示。

同样，我们可以查看 Prometheus 识别的其他一些指标，例如主机状态。下图显示了特定时间段内的主机状态。

Prometheus 中的每个部分的用途如下。

- **AlertManager：**该组件将帮助你根据指标设置服务器的警报并定义其阈值。需要在服务器上添加配置来设置警报。访问 `https://prometheus.io/docs/alerting/alertmanager/`查看有关 AlertManager 的文档。

- **Node exporter**：此 exporter 对于硬件和操作系统指标非常有用。访问 `https://prometheus.io/docs/instrumenting/exporters/`可了解有关不同类型 exporter 的更多信息。
- **Pushgateway**：此 Pushgateway 允许你运行批处理作业来公开你的指标。
- **Grafana**：Prometheus 与 Grafana 集成在一起，可通过仪表板查询 Prometheus 上的指标。

本章小结

本章的内容非常有趣。我们从基于云平台的 CloudWatch 和 Application Insights 等工具出发，在云平台上管理应用程序。然后，转向开源工具，我们知道开源工具一直是开发人员的首选，因为可以根据自己的需求对工具进行定制。我们研究了 ELK 技术栈，这个技术栈一直很受欢迎，有很多组织都在使用它。

现在，已经到了本书的结尾，但笔者还是希望能有一个讨论高级应用程序开发，并附有更多测试用例的版本，那样将对 QA 用户非常有用。最后，享受编程的乐趣吧！